



Universidad de Alcalá

ESCUELA POLITÉCNICA SUPERIOR

MÁSTER UNIVERSITARIO EN INGENIERÍA DE
TELECOMUNICACIÓN

PRÁCTICA ENTREGABLE 1
TRANSMISOR 16 QAM

Diseño de Circuitos Electrónicos para Comunicaciones

Autor:

Paula Bartolomé Mora

17 de enero de 2024

Índice general

1. Introducción	1
2. Captura XADC y memoria FIFO	3
2.1. Obtención de los datos convertidos del ADC	4
2.1.1. Configuración del XADC	4
2.1.2. Simulación de la señal de entrada	4
2.2. Lógica de control de lectura y escritura de los datos de la FIFO	6
2.2.1. Configuración del bloque FIFO	6
2.2.2. Comprobación del bloque FIFO	8
3. Modulador 16-QAM	9
3.1. Mapeado 16-QAM	10
3.1.1. Configuración del mapeado 16-QAM	10
3.2. Proceso de mapeado + Zero Padding	12
3.2.1. Configuración del proceso de mapeado + Zero Padding	12
3.2.2. Comprobación del proceso de mapeado + Zero Padding	13
3.3. Bloque de filtrado pulse shaping - RRC	17
3.3.1. Configuración del filtro RRC	17
3.3.2. Comprobación del filtrado RRC	20
4. Mezclador de Transmisión	22
4.1. Generación de las sinusoides	23
4.1.1. Configuración del DDS	23
4.1.2. Comprobación del DDS	24
4.2. Generación de las componentes I/Q y suma	24
4.2.1. Configuración del proceso de multiplicación y suma	24
4.2.2. Comprobación del proceso de multiplicación y suma	25

5. Generación de relojes	29
6. Conclusiones	31

Capítulo 1

Introducción

El presente trabajo tiene como objetivo principal el desarrollo de un sistema de comunicaciones basado en 16-QAM (Modulación de Amplitud en Cuadratura de 16 niveles). Para ello, se propone el empleo en conjunto de los diferentes bloques electrónicos IP estudiados en la asignatura y se divide el desarrollo del entregable en una serie de fases de configuración:

- **Captura XADC y Memoria FIFO:** En esta primera etapa se realizará la captura de una señal triangular y se procederán a almacenar los datos de entrada en una memoria FIFO.
- **Mapeado QAM:** Se generará el mapeado de 16-QAM para obtener los símbolos a partir de los valores almacenados en la memoria FIFO.
- **Zero Padding:** Se implementará un upscaling o Zero-Padding con relación 1:32 previo al filtrado para conseguir una respuesta de frecuencia óptima y minimizar la interferencias entre símbolos.
- **Filtrado Root Raised Cosine:** Se deberá generar y aplicar a cada rama I/Q el filtrado pulse shaping del *Root Raised Cosine* (RRC).
- **Mezclador DDS y Multiplicadores:** Se generarán las componentes de transmisión de cada rama I/Q a través de la multiplicación de los valores filtrados por funciones coseno/-seno respectivamente. Estas funciones serán obtenidas a partir de un bloque sintetizador de señal (DDS).
- **Sumador:** Se incluirá un bloque final para la suma de las componentes de transmisión de los caminos I y Q.

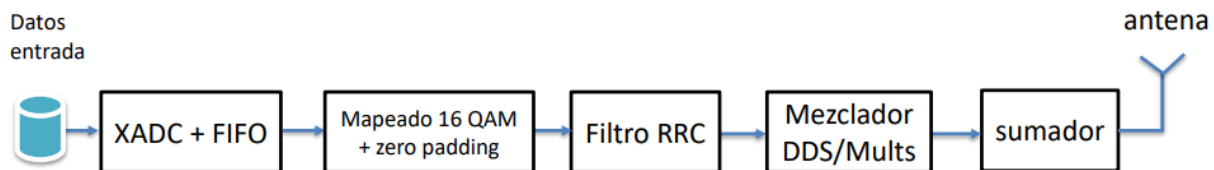


Figura 1.1: Distribución de bloques del sistema de comunicaciones (I)

Para lograr tanto un correcto funcionamiento de cada bloque electrónico como la interoperabilidad entre ellos al integrarlos en el sistema, será preciso un estudio en profundidad de la documentación proporcionada por el fabricante y un análisis cuantitativo y cualitativo de cada uno de los resultados obtenidos en las simulaciones funcionales.

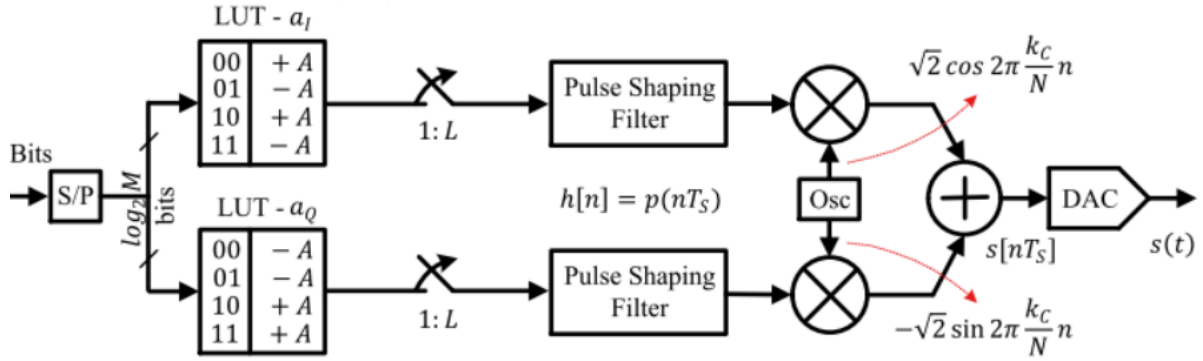


Figura 1.2: Distribución de bloques del sistema de comunicaciones (II)

Como paso adicional de esta práctica y, en relación a la generación de relojes, se creará un uso práctico de un bloque MMCM.

Capítulo 2

Captura XADC y memoria FIFO

Como se ha introducido anteriormente, en este capítulo se definirá el proceso de captura y conversión de una señal de entrada triangular y el posterior almacenamiento de la misma en una memoria FIFO. En la Figura 2.5 se muestra el diseño de los bloques necesarios para esta fase y la definición de sus correspondientes puertos de entrada y salida.

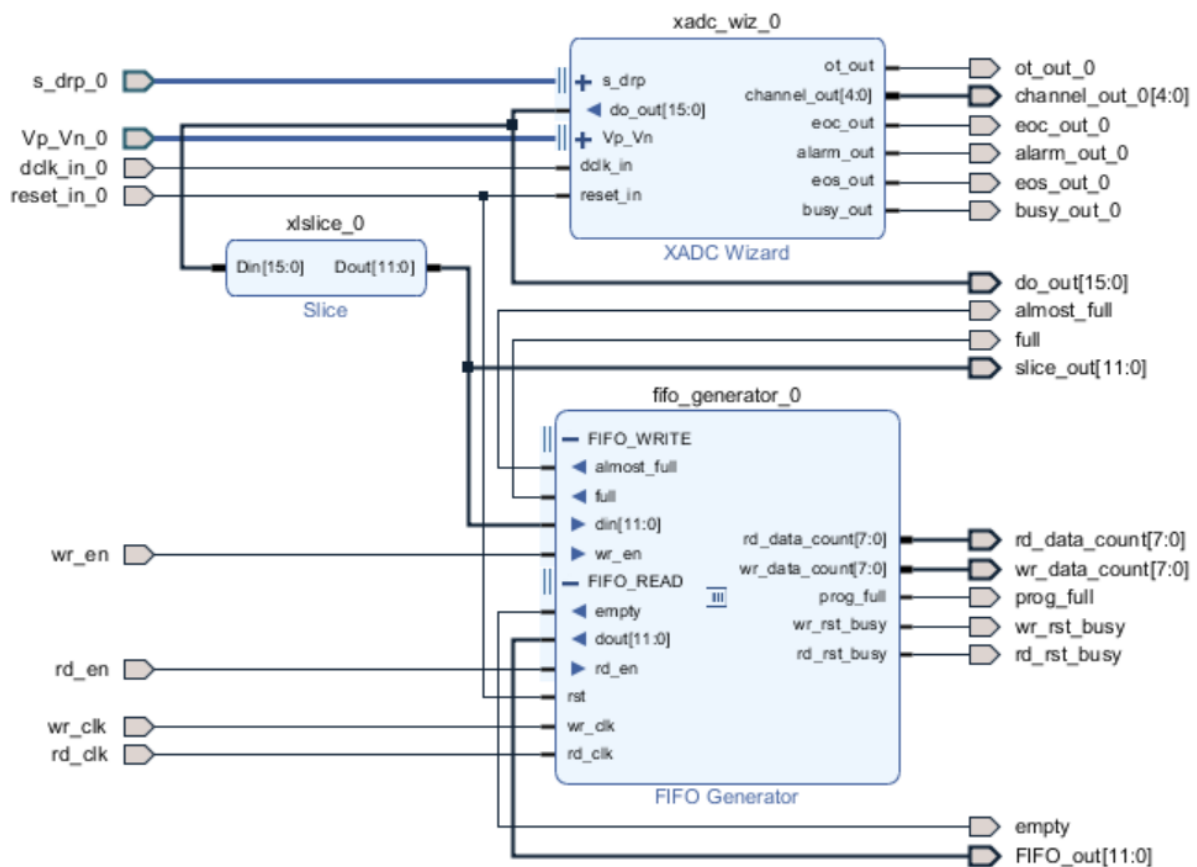


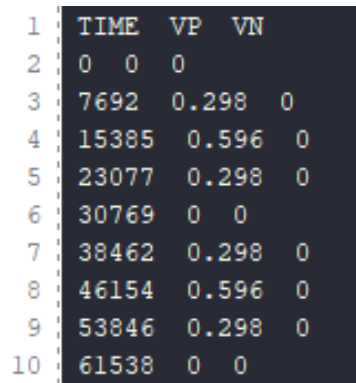
Figura 2.1: Diseño del bloque XADC+FIFO

2.1. Obtención de los datos convertidos del ADC

2.1.1. Configuración del XADC

En primer lugar, teniendo en cuenta que mi puesto de laboratorio es el 8, se debe generar a la entrada una señal triangular con un valor de frecuencia igual a 13KHz y que opere en un rango de valores de tensión entre 0 y 0.596 voltios. Los valores que toma en cada instante temporal serán definidos en el fichero `design.txt`, el cual tendrá que ser importado al proyecto para llevar a cabo la conversión.

Como se puede ver en la Figura 2.2, se definen valores de tensión para dos entradas (VP y VN), que corresponden a las entradas analógicas diferenciales del XADC. Por simplificación, se configura el valor absoluto de tensión para la entrada positiva (VP) y un valor nulo para la negativa (VN).



	TIME	VP	VN
1			
2	0	0	0
3	7692	0.298	0
4	15385	0.596	0
5	23077	0.298	0
6	30769	0	0
7	38462	0.298	0
8	46154	0.596	0
9	53846	0.298	0
10	61538	0	0

Figura 2.2: Valores de tensión (V) de la señal triangular extraídos de `design.txt`

El XADC se configura en modo continuo y con un único canal de entrada para capturar las muestras, por lo que se ha seleccionado el canal 3 para registrar el valor convertido de tensión (`s_drp_daddr`). La conversión se realizará a una velocidad de 1 MS/seg con un reloj de 52 MHz y se establecerá un tiempo de adquisición igual a 4 para que la señal de captura sea estable.

2.1.2. Simulación de la señal de entrada

Una vez configurado el bloque IP XADC se procede a simular la conversión de la señal de entrada. En la figura 2.3 se puede visualizar cómo se obtiene la salida digital de 16 bits por el puerto `do_out`.

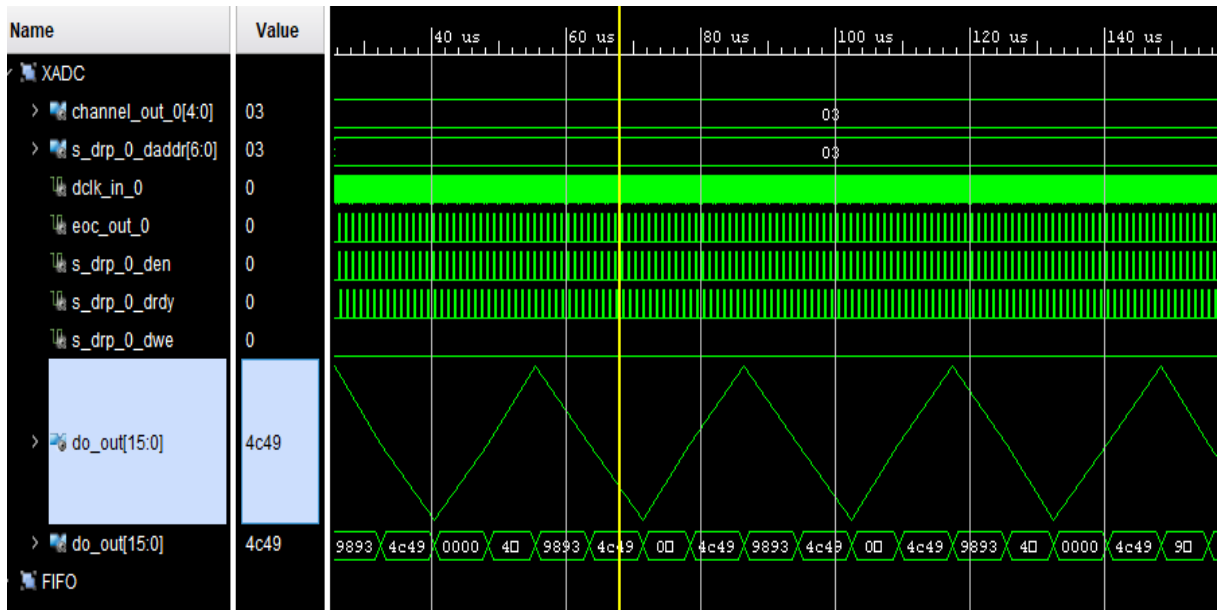


Figura 2.3: Conversión analógico-digital de la señal triangular de entrada

Por otro lado, en la Figura 2.4 se aprecia de una forma más clara los valores que toman cada uno de los puertos del XADC en cada conversión. La señal de fin de conversión *eoc_out* se activa y como consecuencia, también lo hará la de enable del Puerto de Reconfiguración Dinámico (DRP) *s_drp_den* para comenzar la conversión del siguiente dato.



Figura 2.4: Fin de conversión y dato preparado

La operación de lectura se completa cuando la señal de dato ready se activa *s_drp_drdy*, indicando que se el dato está preparado a la salida.

```

1 -- frec. conversion aprox 1MS -> frec 52 KHz (modo continuo)
2 for i in 0 to 200 -- a la espera de 200 datos
3 loop
4     -- End of Conversion signal
5     wait until eoc_out_0 <= '1' ;
6     -- Data ready signal for the dynamic reconfiguration port
7     wait until s_drp_0_drdy = '1';
8 end loop;
```

Listing 2.1: Bucle de conversión de los 200 datos

2.2. Lógica de control de lectura y escritura de los datos de la FIFO

2.2.1. Configuración del bloque FIFO

El bloque de memoria FIFO almacenará cada uno de los datos capturados por el bloque XADC. Consta de una capacidad de 256 posiciones de 12 bits cada una, por lo que a la entrada del bloque FIFO será precisa la configuración de un bloque auxiliar *Slice* para transformar los 16 bits de la señal de salida del XADC en sus 12 bits de mayor peso.

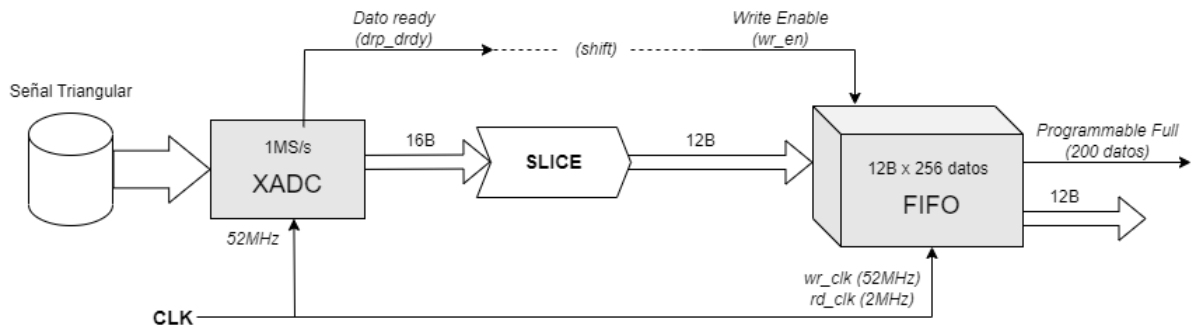


Figura 2.5: Diseño del bloque XADC+FIFO

Se trata de un paso necesario porque el XADC en el fondo maneja una precisión de 12 bits a la salida. En la Figura 2.5 se puede apreciar esta conversión, además de la estructura completa del bloque a implementar.

```

1 process(wr_clk)
2 variable reg : std_logic; -- variable para almacenar dato de salida del
   xadc
3 begin
4 if rising_edge(wr_clk) then
5     if reset_in_0 = '1' then -- reset XADC = 1
6         reg := '0';
7         -- Enable signal for the dynamic reconfiguration port
8         s_drp_0_den <= '0';
9     else
10        s_drp_0_den <= reg ; -- ENABLE = 1 solo durante un ciclo
11        reg := eoc_out_0; -- Se habilita la salida de un nuevo dato
12    end if;
13 end if;
14 end process;
15
16 wr_en <= s_drp_0_drdy; -- se escribe en la FIFO nuevo dato

```

Listing 2.2: Proceso de escritura

Entrando en el proceso de escritura, el funcionamiento sería el siguiente: cada vez que exista un nuevo dato proporcionado por el XADC se escribirá en la memoria FIFO de forma continua a 1 MS/s y con un reloj de 52 MHz (*wr_clk*) hasta llegar a la capacidad máxima.

Una vez la memoria FIFO este llena, comenzará el proceso de lectura de 200 datos a una frecuencia de lectura de 2MHz (*rd_clk*) mediante la activación de la señal (*rd_enable*). En este caso, al desear leer un número de datos menor a la capacidad máxima (256), será necesario configurar una señal auxiliar programable (*programmable_full*).

```

1 process(rd_clk)
2 file outfile: text is out "FIFO_output.txt";
3 variable line_num: line;
4 variable opened, finished: std_logic := '0';
5 begin
6     file_open (outfile, "FIFO_output.txt", write_mode);
7     if rising_edge(rd_clk) then
8         if reset_in_0 = '1' then -- reset XADC = 1
9             rd_en <= '0';
10        else
11            --se comienza a leer la FIFO
12            if prog_full = '1' then
13                rd_en <= '1';
14            end if;
15            if empty = '1' then
16                rd_en <= '0';
17            end if;
18            --proceso de escritura del .txt
19            if rd_en = '1' and finished = '0' then
20                write (line_num, FIFO_out);
21                writeline(outfile, line_num);
22                opened := '1';
23            end if;
24            --Se finaliza la escritura de .txt tras el primer empty
25            if opened = '1' and empty = '1' then
26                finished := '1';
27            end if;
28        end if;
29    end if;
30 end process;

```

Listing 2.3: Proceso de lectura

Adicionalmente, es necesario implementar la escritura de un fichero de salida con los datos almacenados en la FIFO, en el cual cada dato de 12 bits será una nueva línea. Este servirá para simular la lectura de la FIFO desde otro proyecto independiente de Vivado, suponiendo de este modo la entrada del siguiente bloque de mapeado. Se debe limitar la escritura a los primeros 200 datos de salida mediante el uso de variables adicionales (*opened*, *finished*).

2.2.2. Comprobación del bloque FIFO

En las Figuras 2.6 y 2.7 se pueden visualizar los procesos de escritura y de lectura en la memoria FIFO como se ha descrito en el apartado anterior. A modo de depuración de que se está produciendo un correcto funcionamiento según los tiempos configurados, se añaden a la simulación dos señales de contadores de ciclos de lectura y de escritura.

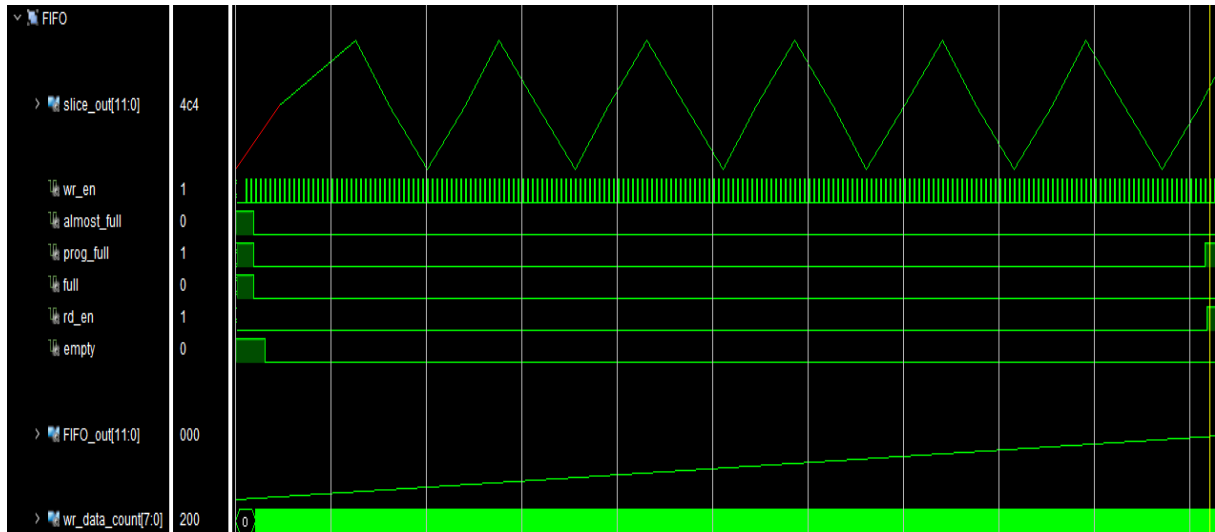


Figura 2.6: Simulación del proceso de escritura en la memoria FIFO

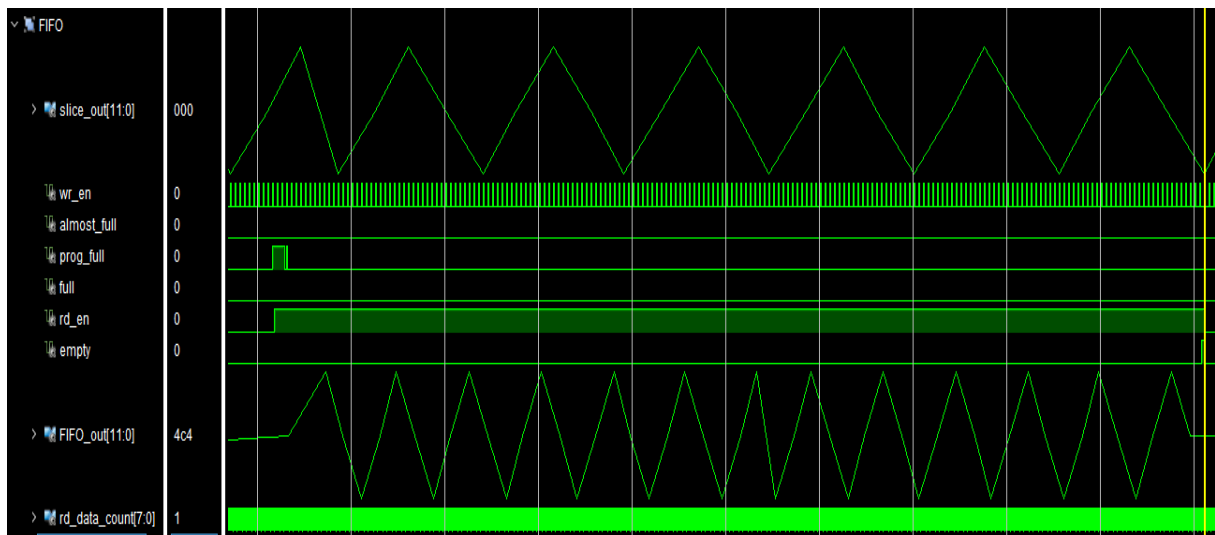


Figura 2.7: Simulación del proceso de lectura en la memoria FIFO

Capítulo 3

Modulador 16-QAM

En este Capítulo se muestra el bloque encargado de la modulación 16-QAM. Este bloque se compone de tres procesos: mapeado 16-QAM, aplicación de Zero Padding y filtrado *Root Raised Cosine*. En la Figura 3.1 se puede visualizar la estructura de los bloques IP que se han incluido en este segundo proyecto de Vivado.

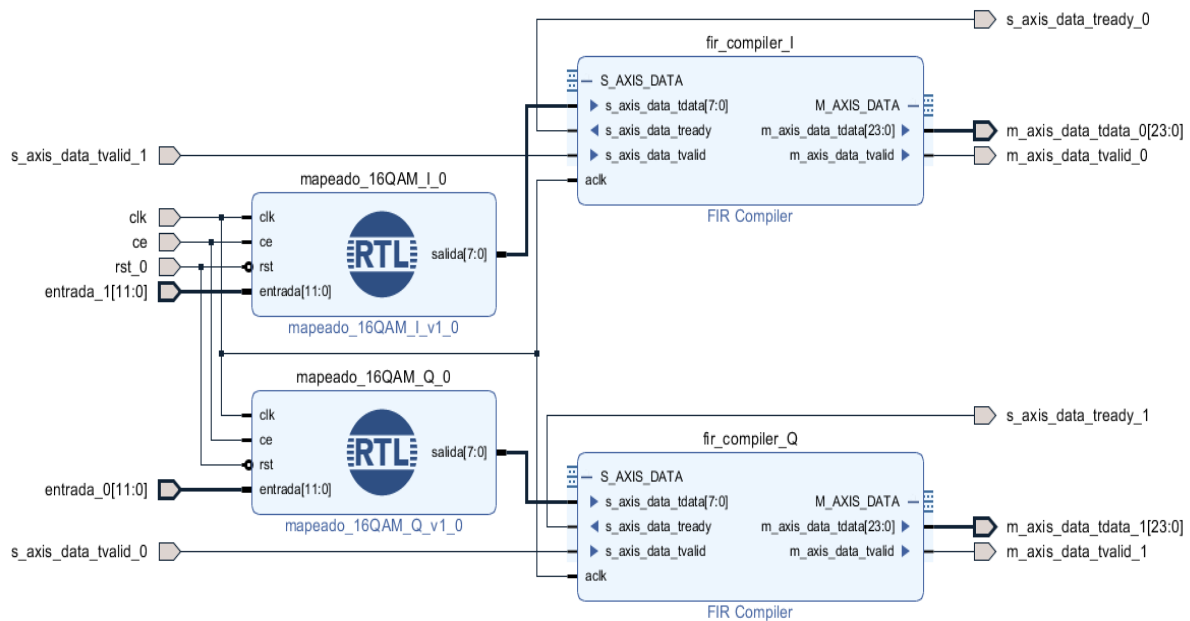


Figura 3.1: Diseño de los bloques mapeado 16-QAM + ZP y filtrado RRC

3.1. Mapeado 16-QAM

3.1.1. Configuración del mapeado 16-QAM

Para implementar una modulación 16-QAM (Modulación de Amplitud en Cuadratura) lo primero que se debe realizar es el bloque de mapeado, con el fin de asignar patrones de amplitud y fase a cada símbolo y proporcionar una mayor eficiencia espectral.

Para ello, se recibe una señal de entrada de 12 bits, obtenida a partir de la lectura de los 200 valores almacenados en la memoria FIFO. Se debe crear un proceso en el testbench que se ejecute en cada ciclo del reloj global, configurado a 192MHz, que realice la lectura del fichero *FIFO_out.txt*. Como se había descrito en el Apartado 2.2.1, se escriben en dicho fichero los valores almacenados en la memoria FIFO para emplearlo como la entrada del bloque de mapeado 16-QAM y optimizar así los proyectos de Vivado.

```
1  process(clk) --192MHz
2      variable line_buffer : line;
3      variable nuevo_valor : STD_LOGIC_VECTOR(11 DOWNT0 0);
4
5      begin
6          if rst_0 = '1' then
7              eof <= false; --se reinicia fin de archivo
8              cont_in <= 0;
9          elsif rising_edge(clk) then
10             cont_in <= cont_in + 1;
11             if cont_in = 96 then --cada 500 ns se lee entrada
12                 cont_in <= 0;
13                 if not eof then
14                     if endfile(file_handle) then
15                         eof <= true;
16                     else
17                         readline(file_handle, line_buffer);
18                         read(line_buffer, nuevo_valor);
19                         entrada <= nuevo_valor;
20                     end if;
21                 end if;
22             end if;
23         end if;
24     end process;
```

Listing 3.1: Proceso de lectura del fichero FIFO_out.txt

Es importante tener en cuenta que la lectura se produce a una frecuencia de 2MHz, por lo que se debe añadir un contador de 96 ciclos en el proceso, además de comprobar si el fichero ha llegado al final. Según las especificaciones de este proyecto, la salida del mapeado debe funcionar a una frecuencia de 6MHz, es decir, el triple de la anterior porque por cada valor a la entrada de 12 bits se obtendrán 3 símbolos 16-QAM. En otros términos, se tratarán los bits de entrada de 4 en 4 para mapear cada símbolo 16-QAM.

El mapeado de cada símbolo 16-QAM supone la generación a la salida de dos caminos independientes: uno para la componente en Fase (I) y otro para la de Cuadratura (Q). Cada símbolo es de 3 bits con signo y puede tomar los valores $(-3, -1, +1, +3)$ como se puede visualizar en la Figura 3.2, procedente del ejemplo simulado en Matlab.

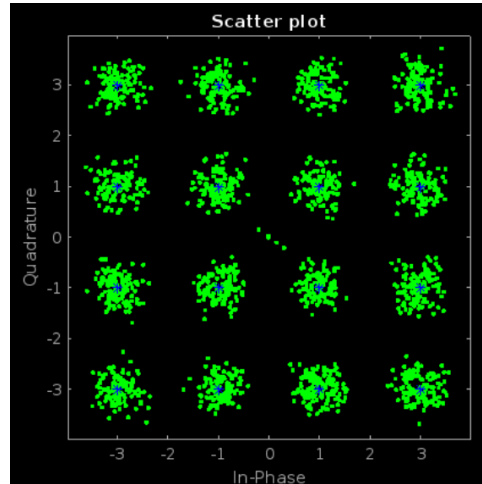


Figura 3.2: Constelación 16-QAM

En este caso, se han decidido ajustar las tablas Q e I para establecer una codificación Gray. Esta se trata de una técnica específica que garantiza que solo un bit cambie entre dos valores consecutivos. Además, consigue minimizar la posibilidad de errores producidos por cambios de múltiples bits en la transmisión. A continuación, se puede visualizar la definición de cada una de las tablas:

```
1 type Array3Bit is array (0 to 15) of STD_LOGIC_VECTOR(2 downto 0);
2 constant tabla_mapeado_Q: Array3Bit :=
3   ("000", "001", "011", "010",
4    "110", "111", "101", "100",
5    "000", "001", "011", "010",
6    "110", "111", "101", "100");
```

Listing 3.2: Definición de la tabla de mapeado Q

```
1 type Array3Bit is array (0 to 15) of STD_LOGIC_VECTOR(2 downto 0);
2 constant tabla_mapeado_I: Array3Bit :=
3   ("000", "000", "001", "001",
4    "011", "011", "010", "010",
5    "110", "110", "111", "111",
6    "101", "101", "100", "100");
```

Listing 3.3: Definición de la tabla de mapeado I

3.2. Proceso de mapeado + Zero Padding

3.2.1. Configuración del proceso de mapeado + Zero Padding

Además de implementar el mapeado 16-QAM, se añade en el bloque el zero-padding para optimizar el sistema de comunicación. Se insertan ceros a la señal original y se extiende su duración, aplicando en este caso una relación 1:32. Por lo tanto, por cada flanco en el que se transmita información, le seguirán 31 flancos en los que se transmitan bits con valor cero.

Siguiendo las especificaciones anteriores, se decide combinar el mapeado 16-QAM con el zero-padding en un mismo bloque, para así obtener a la salida del mismo los símbolos 16-QAM con el Zero Padding ya aplicado. Así, se garantiza que la señal tenga una duración suficiente para conseguir una respuesta de frecuencia óptima, además de minimizar la interferencia entre símbolos. De forma adicional, se ajusta la salida a una longitud de 8 bits para dejarla preparada para la entrada del filtro pulse shaping del *Root Raised Cosine* (RRC).

```
1 process(clk)
2   variable contador : integer := 0; -- 4*contador+3 downto 4*contador
3   variable cont_32 : integer := 0; -- 6MHz
4   variable captured_bits : std_logic_vector(3 downto 0) := "0000";
5   variable aux_salida : std_logic_vector(2 downto 0) := "000";
6 begin
7   if rising_edge(clk) then
8     if rst = '1' then
9       contador := 0;
10      cont_32 := 0;
11      salida <= (others => '0');
12    else
13      if cont_32 = 32 then -- se saca simbolo por la salida
14        cont_32 := 0;
15        captured_bits := entrada(4*contador+3 downto 4*contador);
16        aux_salida := tabla_mapeado_I(to_integer(unsigned(captured_bits)
17      ));
18      --salida extendida a 8 bits
19      salida(2 downto 0) <= aux_salida;
20      salida(7 downto 3) <= (others => aux_salida(2));
21      contador := contador + 1;
22      if contador = 3 then
23        contador := 0;
24      end if;
25    else --Zero Padding
26      cont_32 := cont_32 + 1;
27      salida <= (others => '0');
28    end if;
29  end if;
30 end process;
```

Listing 3.4: Proceso de mapeado (Camino I) + Zero Padding

Como se puede visualizar, el proceso emplea dos contadores: uno para extraer en cada flanco los bits capturados de 4 en 4 e implementar el mapeado QAM para obtener los símbolos de 3 bits y otro, para implementar la relación 1:32 respectiva al Zero Padding.

3.2.2. Comprobación del proceso de mapeado + Zero Padding

En el testbench se incluye el proceso de estimulación de las señales de habilitación (*clock enable*) y de reset del bloque de mapeado. También, se pueden apreciar las líneas de código dedicadas a la activación de las señales de validación de datos del filtro (*s_axis_data_tvalid*), pero esto se detallará en el Apartado 3.3.2.

Además, se deben abrir los ficheros correspondientes a la lectura y escritura de datos para cada uno de los bloques: para lectura de las muestras almacenadas en la FIFO y para escritura de las salidas de las ramas de filtrado.

```
1 process
2 begin
3     file_open(file_handle, ".\FIFO_output.txt", READ_MODE);
4     file_open(file_handle_I, ".\FIR_output_I.txt", WRITE_MODE);
5     file_open(file_handle_Q, ".\FIR_output_Q.txt", WRITE_MODE);
6
7     ce <= '0';
8     rst_0 <= '1';
9
10    s_axis_data_tvalid_0 <= '0';
11    s_axis_data_tvalid_1 <= '0';
12
13    wait for 32*clk_period;
14    rst_0 <= '0';
15
16    wait for 32*clk_period;
17    ce <= '1';
18    s_axis_data_tvalid_0 <= '1';
19    s_axis_data_tvalid_1 <= '1';
20
21    wait;
22 end process;
```

Listing 3.5: Proceso de estimulación de señales de reset y clock enable

A continuación, en las Figuras 3.3, 3.4 y 3.5 se puede apreciar cómo se implementa el bloque de mapeado 16-QAM + zero-padding a través del proceso expuesto anteriormente.

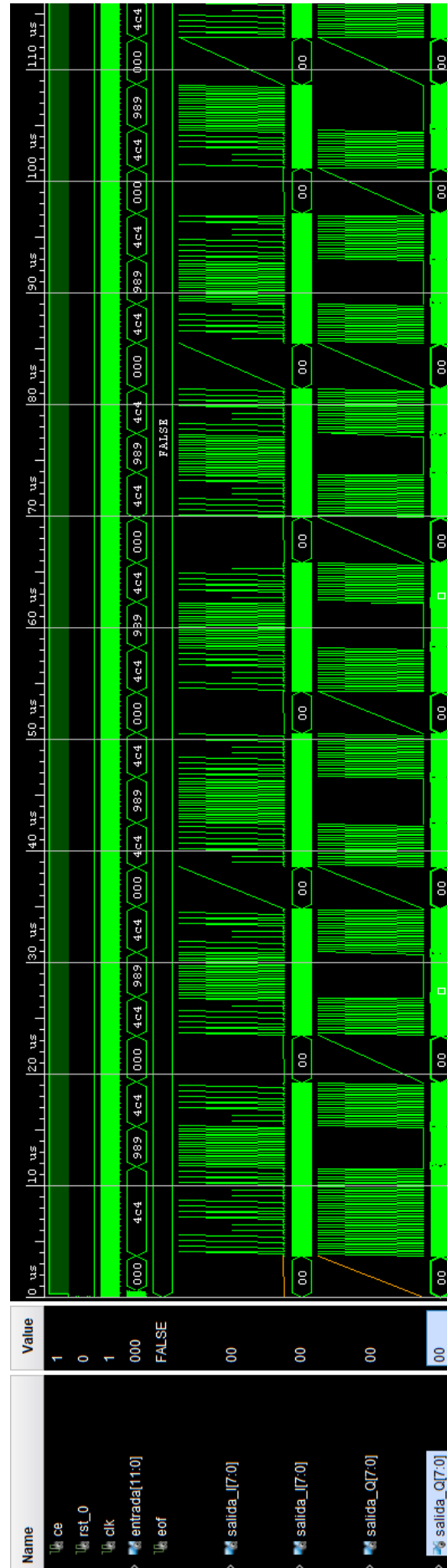


Figura 3.3: Simulación completa del proceso de mapeado 16-QAM + ZP

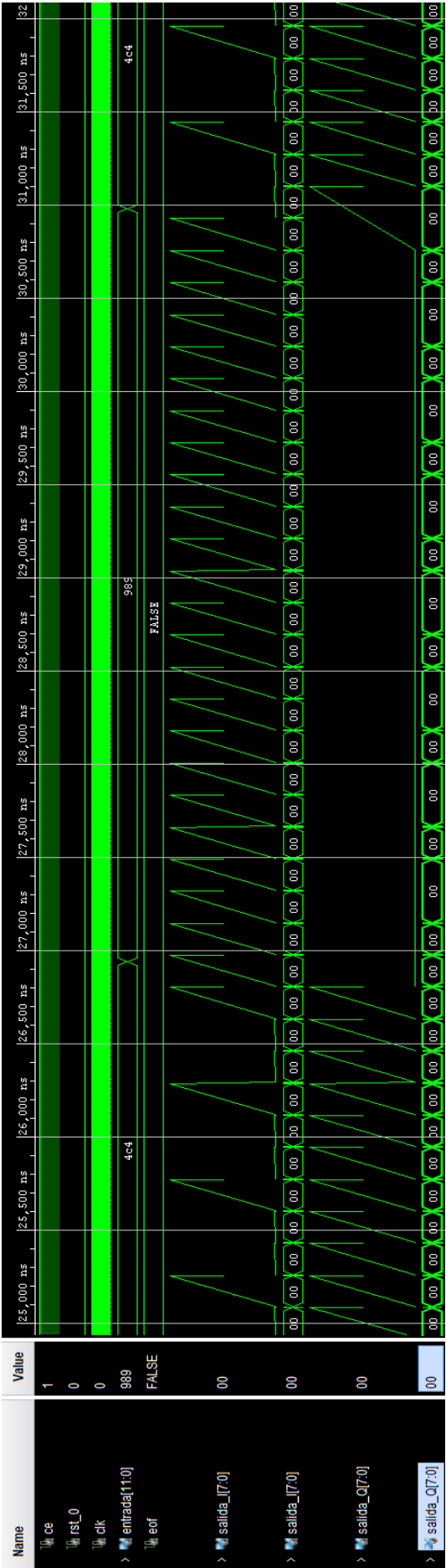


Figura 3.4: Simulación del proceso de mapeado 16-QAM + ZP (II) (zoom medio)

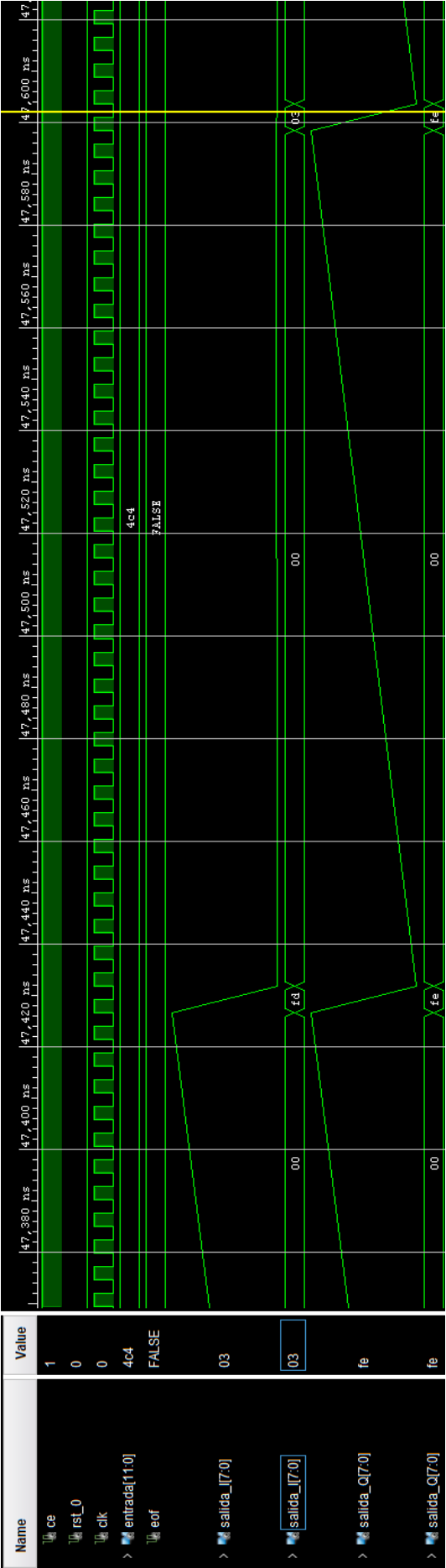


Figura 3.5: Simulación del proceso de mapeado 16-QAM + ZP (III) (zoom alto)

3.3. Bloque de filtrado pulse shaping - RRC

3.3.1. Configuración del filtro RRC

El filtro pulse shaping del *Root Raised Cosine* (RRC) es un tipo de filtro denominado de coseno elevado. Se emplea en sistemas de comunicación digital para dar forma de onda a los pulsos que se transmiten. Minimiza la interferencia entre símbolos adyacentes y maximiza la eficiencia del espectro.

El primer paso a seguir para crear el filtro RRC es obtener los coeficientes del mismo. Para ello, se va a emplear Matlab, estableciendo de la siguiente forma las especificaciones del filtro a diseñar:

```
1 filtlen = 6;           % Longitud del filtro en numero de simbolos
2 rolloff = 0.25;       % Factor de rolloff para controlar el ancho del pulso
3 sps = 32;             % Muestras por simbolo (factor de oversampling)
4
5 rrcFilter = rcosdesign(rolloff, filtlen, sps);
6 fvtool(rrcFilter, 'Analysis', 'Impulse')
```

Listing 3.6: Diseño del filtro RRC en Matlab

Tras ejecutar el código, se obtendrán en total 193 coeficientes. Esto tiene como consecuencia que, para aplicar el filtro a la señal de entrada se irá ajustando un enventanado de longitud igual a 193 datos (dato actual + 192 datos siguientes) hasta finalizar la etapa.

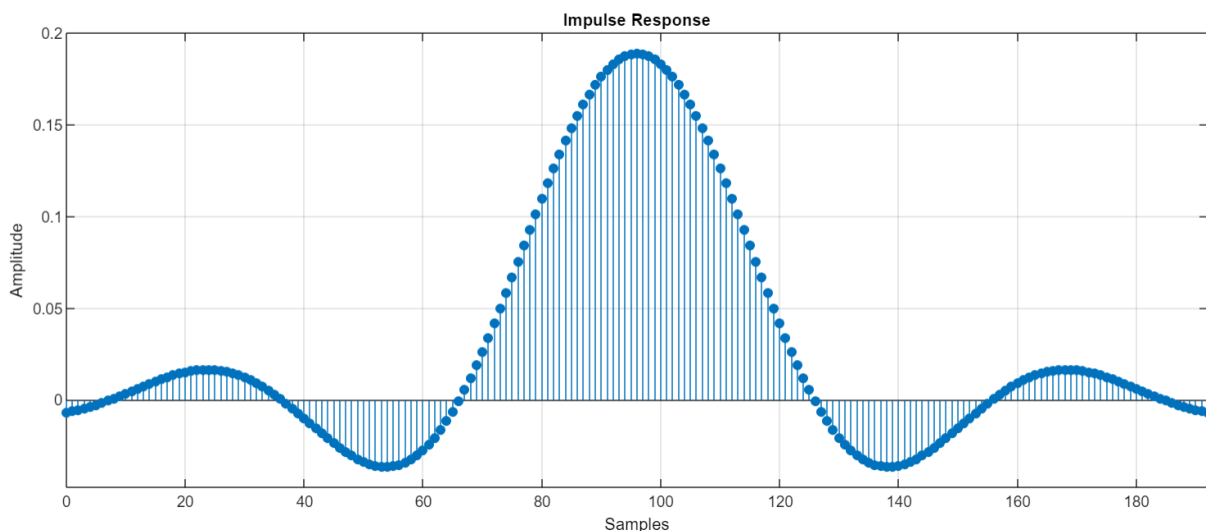


Figura 3.6: Respuesta al impulso del filtro RRC

Mediante la herramienta gráfica *fvtool* que proporciona Matlab se puede obtener la respuesta impulsiva del filtro RRC tal y como se muestra en la Figura 3.6.

Para añadir el bloque del filtro en Vivado se requiere conocer el número de bits de los coeficientes, tanto de la parte entera como de la parte fraccionaria. Para ello, se emplea el comando de MatLab *sfi(rrcFilter)* y se obtiene la salida de la Figura 3.7

```

DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 16
FractionLength: 17

```

Figura 3.7: Información sobre la matriz de coeficientes del filtro RRC

Como los coeficientes se encuentran dentro de un rango de valores entre 0 y 1, se establecen 16 bits para la logitud total y 17 bits, para la parte fraccionaria. Por otro lado, se configura el modo de truncamiento, una salida de filtrado de 24 bits y una entrada de 8 bits, que ya ha sido previamente ajustada en el bloque anterior, por lo que no se requieren procesos adicionales (ver Figura 3.8).

Coefficient Options

Coefficient Type Signed

Quantization Quantize Only

Coefficient Width 16 [2 - 49]

☒ Best Precision Fraction Length

Coefficient Fractional Bits 17 [0 - 17]

Coefficient Structure

☐ Inferred

☐ Non Symmetric

☒ Symmetric

Data Path Options

Input Data Type Signed

Input Data Width 8 [2 - 47]

Input Data Fractional Bits 0 [0 - 8]

Output Rounding Mode Truncate LSBs

Output Width 24 [1 - 28]

Output Fractional Bits : 12

Figura 3.8: Configuración del filtro en Vivado

Entrando en el código, para que el filtrado comience a operar es necesario primero resetear las entradas de validación de datos *s_axis_data_tvalid* de ambos filtros (camino I y Q) y después, activarlas cuando la señal de *clock enable* se pone a 1.

Por otro lado, como se había realizado con la salida de la memoria FIFO, para el bloque de filtrado también se van a almacenar los valores de salida en un fichero .txt. Para ello, se emplea el proceso definido anteriormente para la lectura de la FIFO y se añaden las líneas necesarias de escritura de los nuevos ficheros.

```

1 process(clk) --192MHz
2   --lectura datos FIFO
3   variable line_buffer : line;
4   variable nuevo_valor : STD_LOGIC_VECTOR(11 DOWNT0 0);
5
6   --escritura datos FIR
7   variable line_buffer_I, line_buffer_Q : line;
8   variable nuevo_valor_I, nuevo_valor_Q : integer;
9
10  begin
11    if rst_0 = '1' then
12      eof <= false; --se reinicia fin de archivo
13      cont_in <= 0;
14    elsif rising_edge(clk) then
15      cont_in <= cont_in + 1;
16      if cont_in = 96 then --cada 500 ns se lee entrada
17        cont_in <= 0;
18        if not eof then
19          if endfile(file_handle) then
20            eof <= true;
21          else
22            readline(file_handle, line_buffer);
23            read(line_buffer, nuevo_valor);
24            entrada <= nuevo_valor;
25          end if;
26        end if;
27      end if;
28      if s_axis_data_tvalid_0 = '1' and s_axis_data_tvalid_1 = '1'
29      and not eof then
30        --escritura de salida FIR I a 192MHz
31        nuevo_valor_I:=to_integer(signed(m_axis_data_tdata_0));
32        write(line_buffer_I, nuevo_valor_I);
33        writeline(file_handle_I, line_buffer_I);
34        --escritura de salida FIR Q a 192MHz
35        nuevo_valor_Q:=to_integer(signed(m_axis_data_tdata_1));
36        write(line_buffer_Q, nuevo_valor_Q);
37        writeline(file_handle_Q, line_buffer_Q);
38      end if;
39    end if;
40  end process;

```

Listing 3.7: Proceso de escritura de los ficheros FIR_output_I.txt y FIR_output_Q.txt

3.3.2. Comprobación del filtrado RRC

En las Figuras 3.9 y 3.10 se visualiza la simulación completa del proyecto que incluye los bloques de mapeado 16-QAM + zero-padding y filtrado RRC. Se puede comprobar cómo el filtro da forma de onda a los pulsos que se transmiten y que provienen del proceso de mapeado previo.

Como se había adelantado en el Apartado 3.2.2, en el testbench se deben activar las señales de validación de datos del filtro (*s_axis_data_tvalid*) a la vez que se produce la habilitación del *clock enable* del mapeado. Esto es imprescindible añadirlo para que no se realice el filtrado si no existen nuevas muestras de entrada.

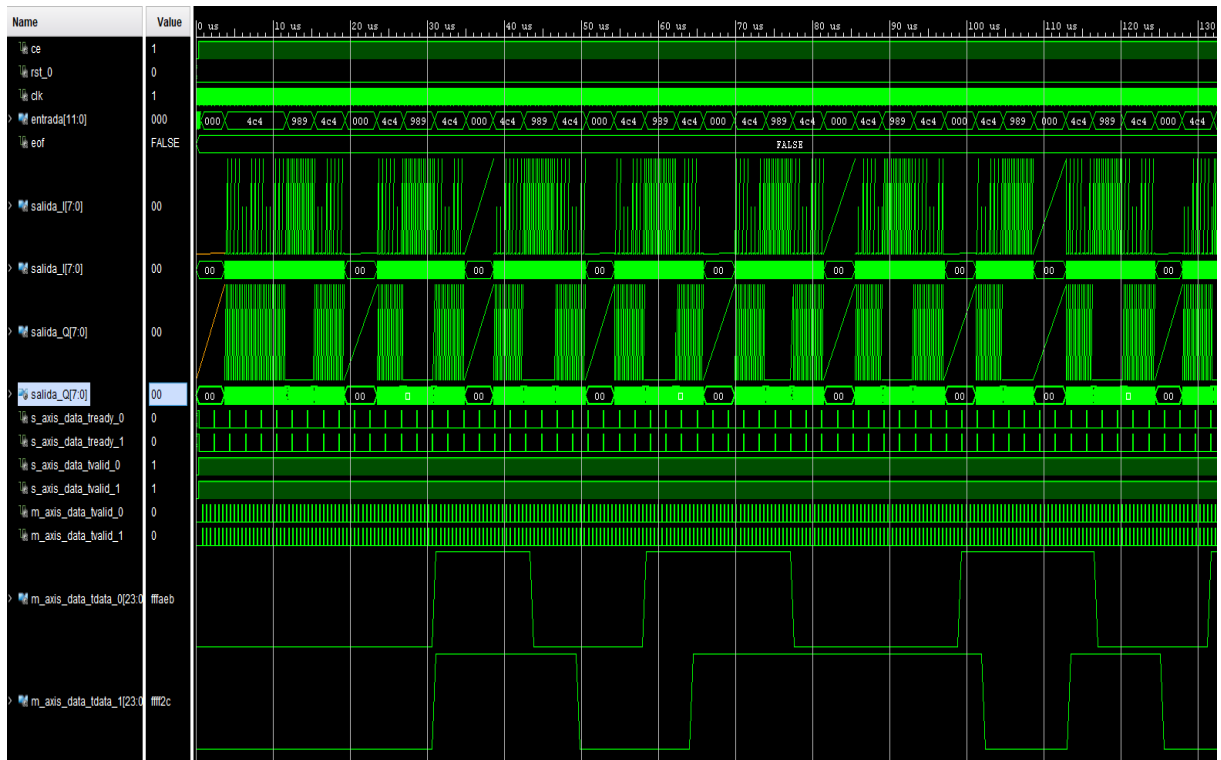


Figura 3.9: Simulación de los bloques de mapeado 16-QAM + ZP y filtrado RRC (I)

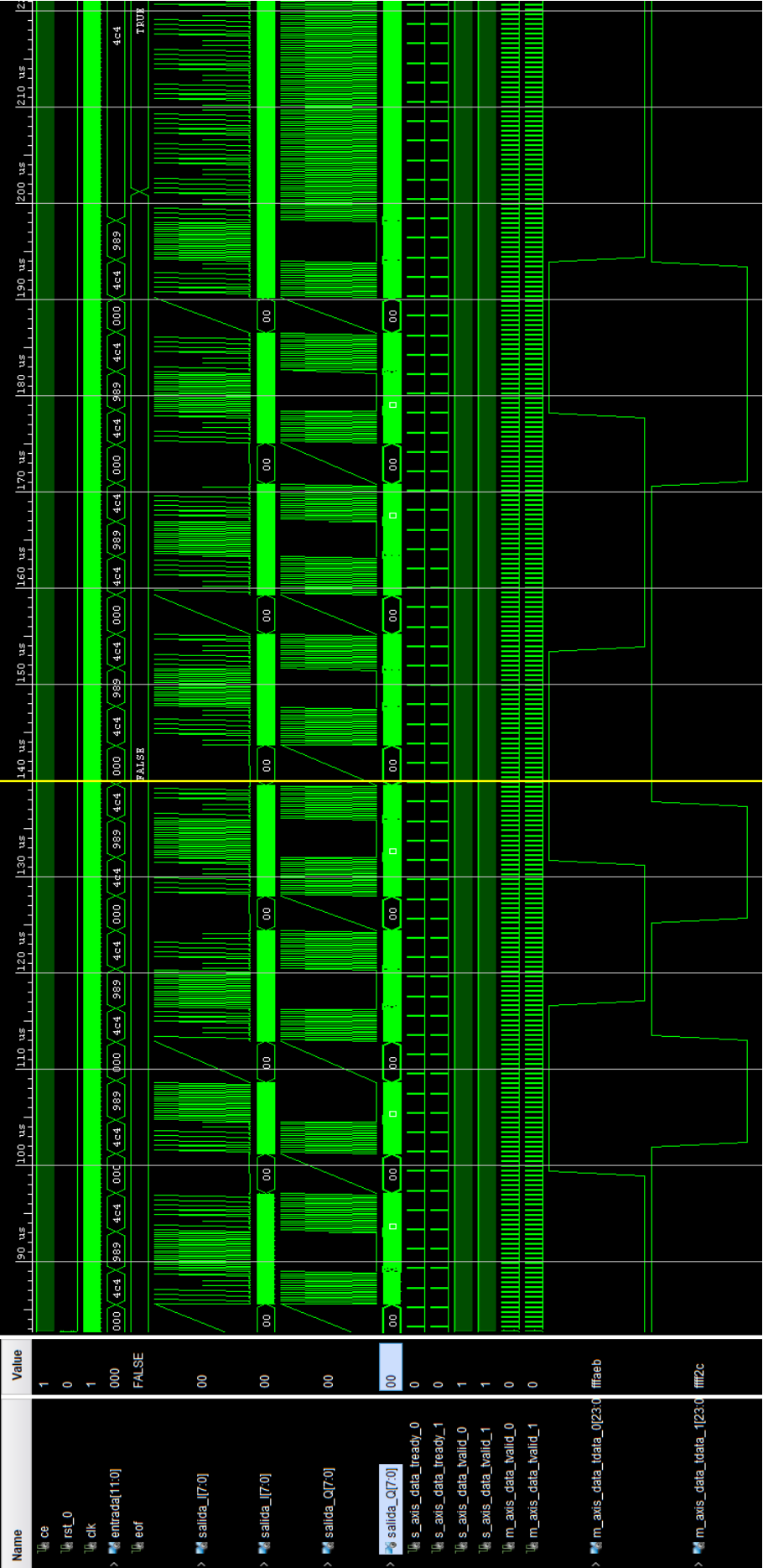


Figura 3.10: Simulación de los bloques de mapeado 16-QAM + ZP y filtrado RRC (II)

Capítulo 4

Mezclador de Transmisión

En este Capítulo se describe el último bloque del sistema, correspondiente al mezclador de transmisión. Este bloque se compone de tres procesos: la generación de señales sinusoidales mediante un sintetizador de señal (DDS), la multiplicación de los datos I/Q por estas señales sinusoidales y la suma final de las componentes I/Q de transmisión. En la Figura 4.1 se visualiza la estructura de los bloques IP que se han incluido en este tercer proyecto de Vivado.

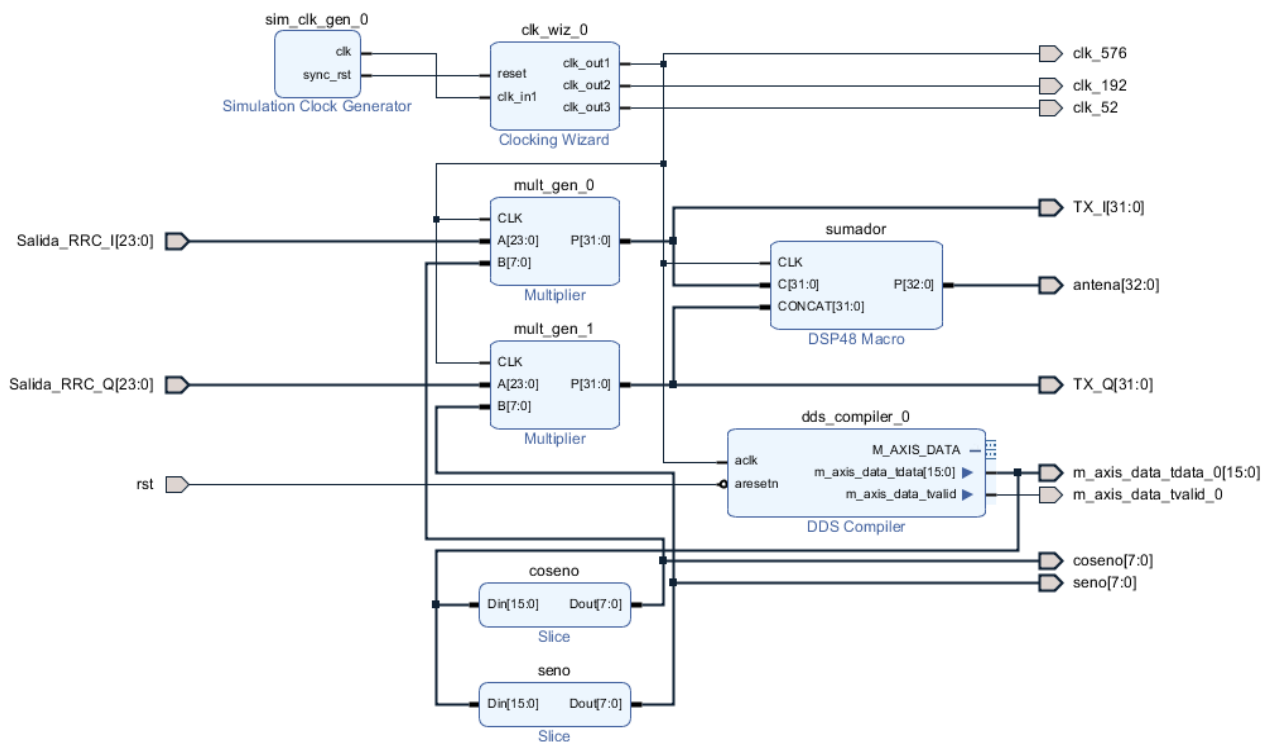


Figura 4.1: Diseño del bloque mezclador de transmisión

4.1. Generación de las sinusoides

4.1.1. Configuración del DDS

Se deben generar dos señales sinusoidales, en particular una función coseno y otra -seno, a partir de un bloque DDS (*Direct Digital Synthesis*). Este bloque es capaz de generar señales analógicas de forma directa mediante un oscilador controlado numéricamente (NCO) y para ello, es preciso configurar internamente la frecuencia de salida deseada de las señales.

En el caso de este proyecto será de 100MHz y el reloj del sistema tendrá una frecuencia de 576MHz (3x192MHz), por lo que se obtendrán 5/6 muestras por ciclo. Como se expondrá en detalle en el Capítulo 5, se incorporará en este proyecto un bloque MMCM para extraer la señal de reloj de 576MHz que se requiere.

La salida del bloque DDS (*m_axis_data_tdata*) está constituida por 16 bits, de los cuales 8 corresponderán a la señal de -seno y los otros 8, a la de coseno. Esto se puede comprobar en la configuración realizada en la Figura 4.3 y en la pestaña de información del bloque, que se muestra en la Figura 4.2. Es por ello que se requiere dividir la salida y definir los bits que se dirigirán a la entrada de cada multiplicador mediante dos bloques *slice*.

The image shows a configuration window for a DDS block. It has three sections: 'Output Selection' with radio buttons for 'Sine', 'Cosine', and 'Sine and Cosine' (which is selected); 'Polarity' with checkboxes for 'Negative Sine' (checked) and 'Negative Cosine'; and 'Amplitude Mode' with a dropdown menu set to 'Full Range'.

Figura 4.2: Configuración de salida de las señales sinusoidales en el bloque DDS

The image shows the 'AXI4-Stream Port Structure' window. It displays a table for 'M_AXIS_DATA - TDATA' with columns 'Transaction', 'Field', and 'Type'. Transaction 0 is shown with two fields: 'CHAN_0_SINE(15:8)' of type 'fix8_7' and 'CHAN_0_COSINE(7:0)' of type 'fix8_7'.

Transaction	Field	Type
0	CHAN_0_SINE(15:8)	fix8_7
	CHAN_0_COSINE(7:0)	fix8_7

Figura 4.3: Información sobre las señales sinusoidales a la salida del bloque DDS

4.1.2. Comprobación del DDS

En la Figura 4.4 se puede visualizar por un lado, la salida de 16 bits obtenida del bloque DDS y, por otro lado, las dos ondas sinusoidales (coseno/-seno) de 8 bits obtenidas a partir de la selección de bits expuesta en la Figura 4.3.

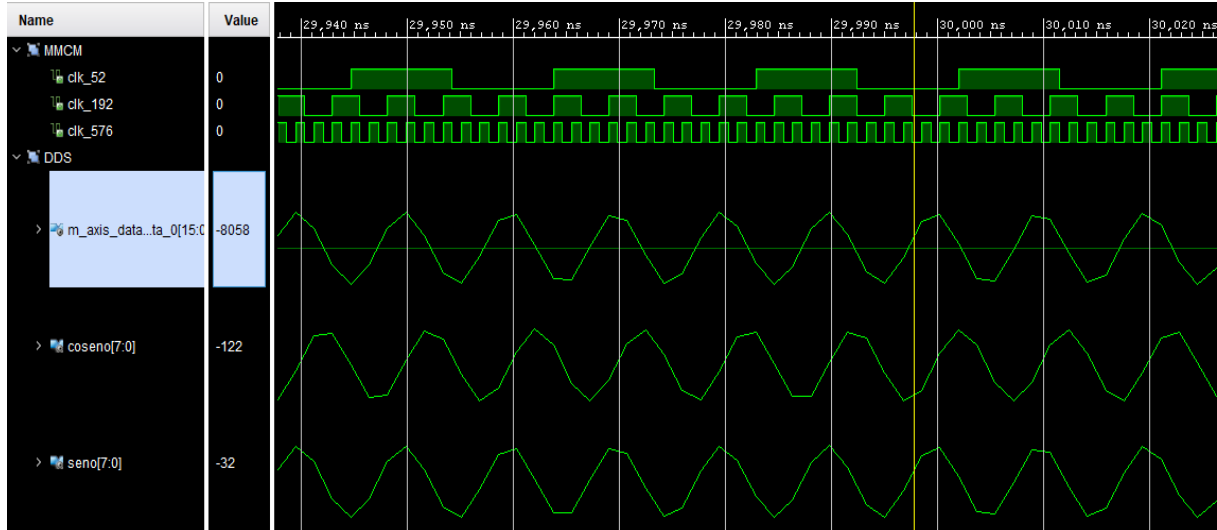


Figura 4.4: Simulación de la salida del bloque DDS (*m_axis_data.tdata*) y de las ondas sinusoidales como resultado de la incorporación de los dos bloques *slice*

4.2. Generación de las componentes I/Q y suma

4.2.1. Configuración del proceso de multiplicación y suma

Para generar las componentes I/Q de transmisión se deben multiplicar (con un bloque IP multiplicador) las salidas I/Q del filtrado realizado anteriormente por las funciones coseno/-seno generadas en el bloque DDS. Como se ha introducido, el procesamiento de los datos del filtro y el bloque de multiplicación trabajarán a una frecuencia de 576 MHz, siendo el triple de la empleada en los bloques anteriores.

Como proceso final, se deben combinar las dos componentes I/Q generadas a la salida de los multiplicadores en un bloque sumador. Este se configurará con dos entradas de 32 bits (C y CONCAT) y una salida de 33 bits.

En cuanto a las salidas I/Q del filtrado (*Salida_RRC_I* y *Salida_RRC_Q*), es preciso extraer los datos a partir de la lectura de los ficheros *FIR_output_I.txt* y *FIR_output_Q.txt*, que fueron generados en el proyecto anterior. A continuación, se muestra el proceso de lectura en el caso de la rama I, siendo análogo para la rama Q.

```

1 process(clk)
2     --lectura datos salidas filtros
3     variable line_buffer : line;
4     variable nuevo_valor : integer;
5
6     begin
7         if rst = '0' then
8             eof_I <= false; --se reinicia fin de archivo
9         elsif rising_edge(clk) then
10             if not eof_I then
11                 if endfile(file_handle_I) then
12                     eof_I <= true;
13                 elsif m_axis_data_tvalid_0 = '1' then
14                     readline(file_handle_I, line_buffer);
15                     read(line_buffer, nuevo_valor);
16                     Salida_RRC_I <= std_logic_vector(to_signed(
nuevo_valor, Salida_RRC_I'length));
17                 end if;
18             end if;
19         end if;
20 end process;

```

Listing 4.1: Proceso de lectura del fichero de salida del filtrado (rama I)

4.2.2. Comprobación del proceso de multiplicación y suma

En el testbench se añade el proceso principal para activar y desactivar la señal de reset, que irá conectada al bloque DDS y de la cual también dependerá el comienzo de lectura de los ficheros de datos del filtro. Es imprescindible configurarlo de esta manera para que se realice tanto la generación de las ondas sinusoidales como la lectura del filtrado de forma simultánea para poder operar.

```

1 process
2 begin
3     file_open(file_handle_I, ".\FIR_output_I.txt", READ_MODE);
4     file_open(file_handle_Q, ".\FIR_output_Q.txt", READ_MODE);
5
6
7     rst <= '0'; -- negativo
8     wait for 100 ns;
9     rst <= '1';
10
11     wait;
12 end process;

```

Listing 4.2: Proceso de estimulación

A continuación, en las Figuras 4.5 y 4.6 se muestran los resultados de la simulación de los bloques multiplicadores tanto para la rama I como para la Q. Las salidas obtenidas de estos bloques multiplicadores (TX_I y TX_Q) supondrán las entradas del bloque sumador. En la Figura 4.7 se puede visualizar cómo se obtiene de la suma la salida final *antena* de 33 bits.

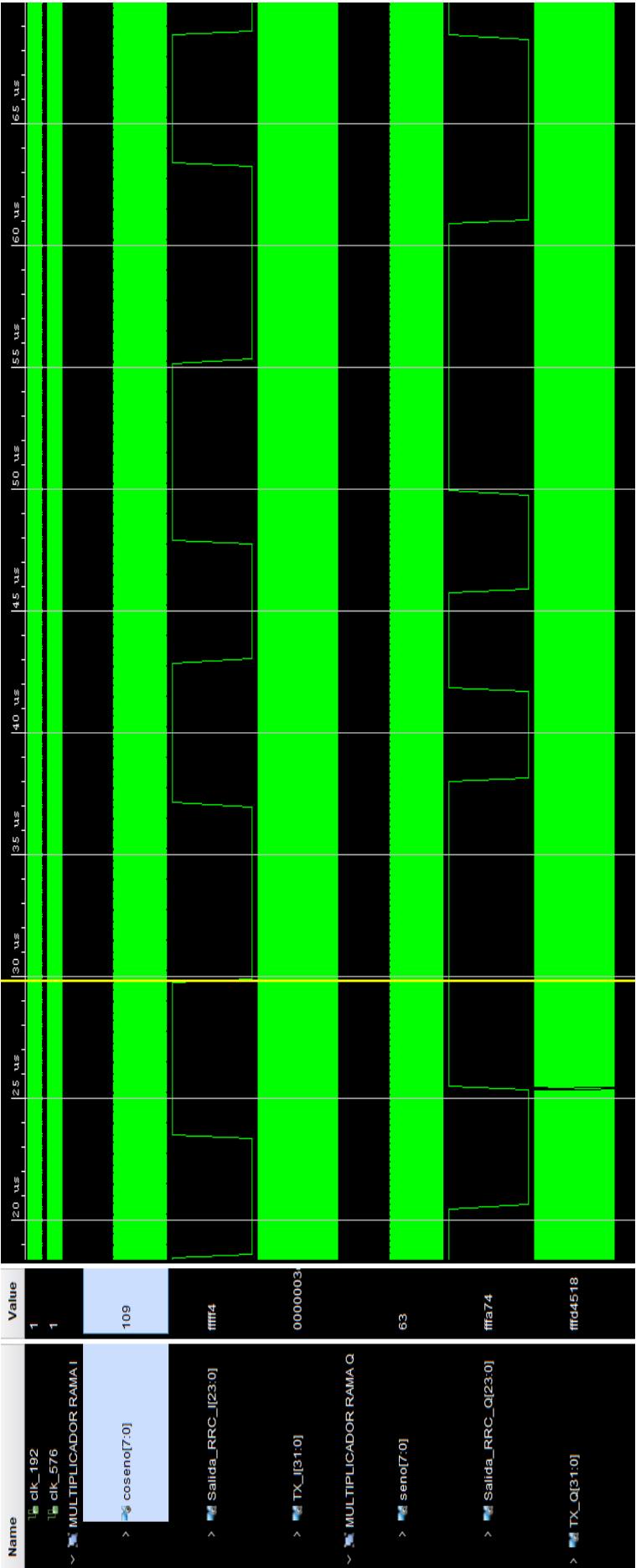


Figura 4.5: Simulación de la operación de multiplicación (I) (sin zoom)

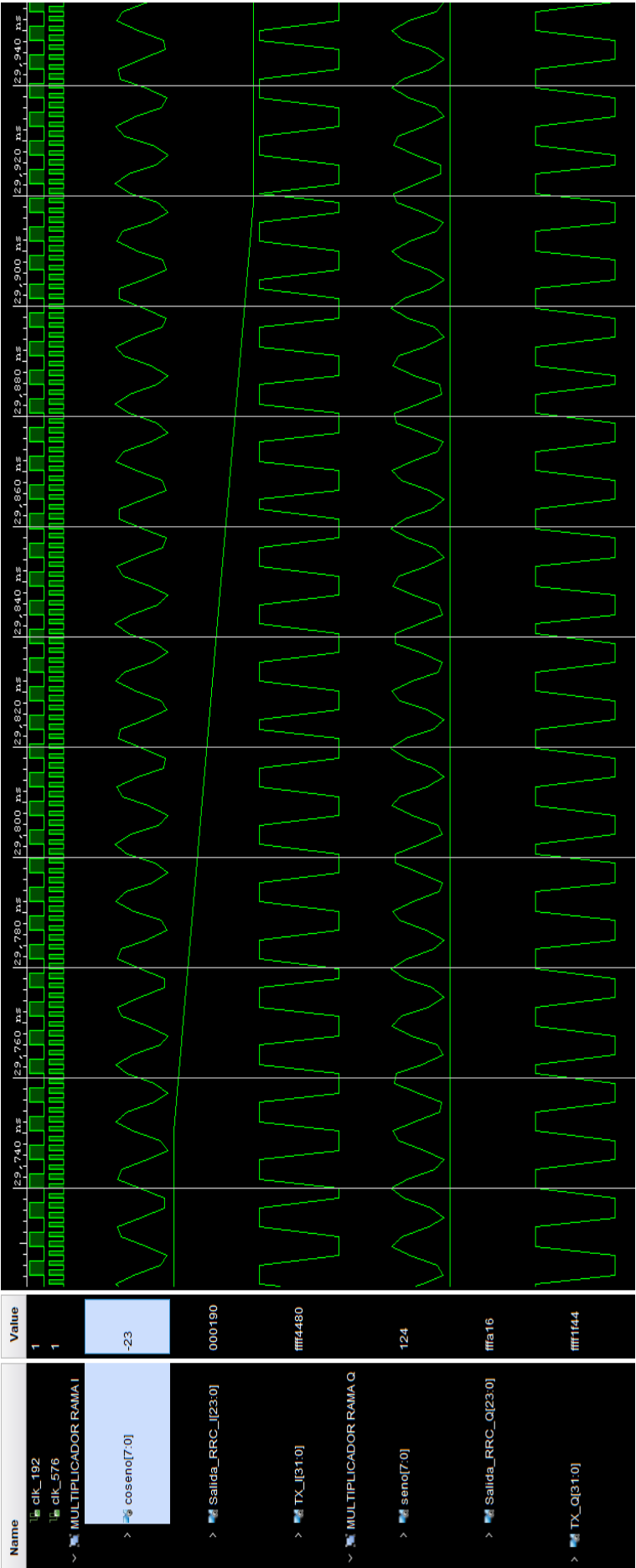


Figura 4.6: Simulación de la operación de multiplicación (II) (con zoom)

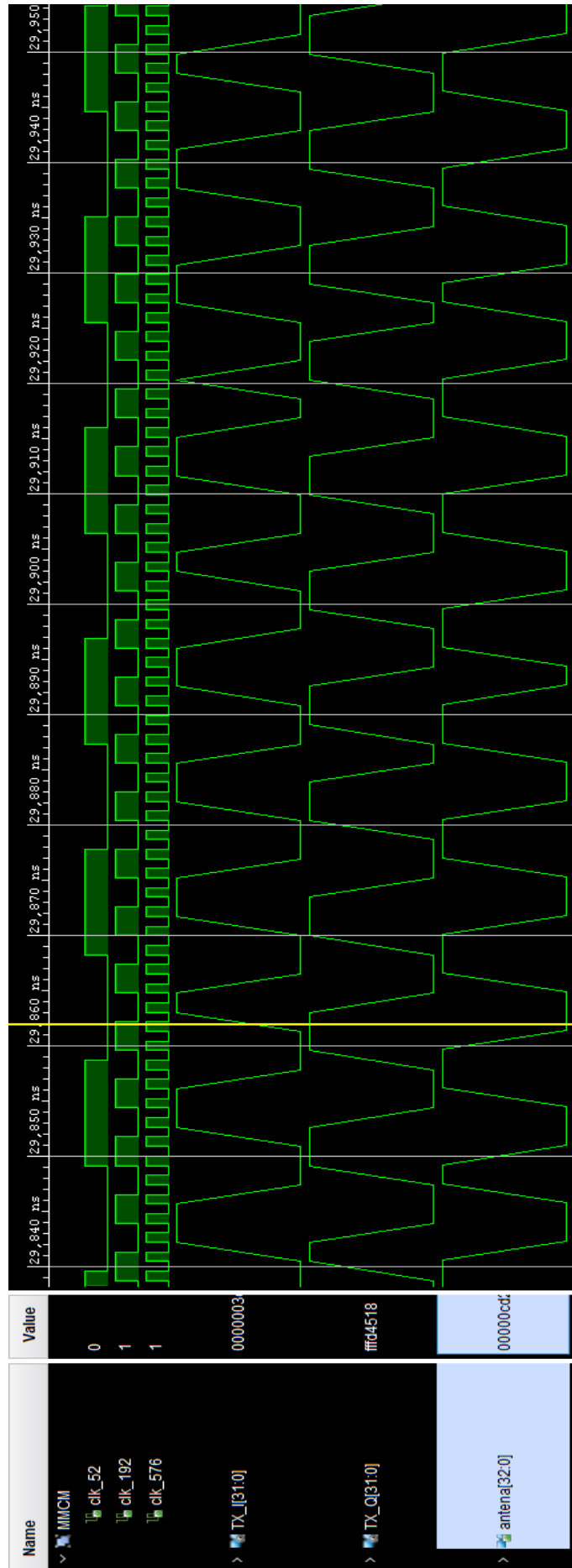


Figura 4.7: Simulación de la operación de suma

Capítulo 5

Generación de relojes

En relación a la generación de señales de reloj, se ha decidido implementar un bloque MMCM en el último proyecto de Vivado, perteneciente al bloque del mezclador (ver Capítulo 4).

El objetivo es hacer un uso práctico de este bloque en uno de los módulos del sistema para validar su funcionamiento, mientras que en el resto se trasladaría directamente la señal de reloj correspondiente. Se considera partir de un reloj externo de frecuencia igual a 36 MHz para generar a la salida tres relojes: de 576 MHz (DDS/Multiplicadores), 192 MHz (RRC) y 52 MHz (XADC).

Para ello, se añaden al proyecto dos bloques IP: *Simulation Clock Generation*, para generar el reloj externo y *Clocking Wizard*, para obtener las tres salidas de reloj con diferentes frecuencias. En la Figura 5.1 se puede observar la configuración del bloque IP para la generación de relojes *Clocking Wizard*.

Primary Input Clock Attributes

Input Clock Frequency (MHz)	36.000
Clock Source	Single_ended_clock_capable_pin
Jitter	0.010

Clocking Primitive Attributes

Primitive Instantiated : MMCM

Divide Counter : 1

Mult Counter : 32.000

Clock Phase Shift : None

Clock Wiz O/p Pins	Source	Divider Value	Tspread (ps)	Pk-to-Pk Jitter (ps)	Phase Error (ps)
clk_out1	MMCM CLKOUT0	2.000	OFF	129.300	173.591
clk_out2	MMCM CLKOUT1	6	OFF	151.084	173.591
clk_out3	MMCM CLKOUT2	22	OFF	193.692	173.591

Figura 5.1: Resumen de configuración del bloque IP *Clocking Wizard*

Como se implementa en el módulo del mezclador, se cogerá la salida con frecuencia 576 MHz y se conectará la misma a todos los bloques.

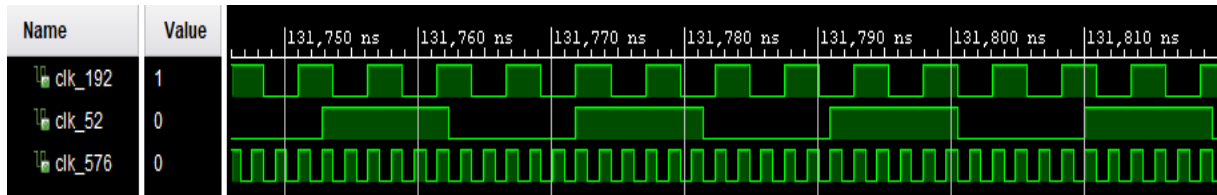


Figura 5.2: Comprobación del funcionamiento del bloque IP *Clocking Wizard*

Capítulo 6

Conclusiones

Con este trabajo se ha conseguido aportar de forma significativa un mayor entendimiento sobre los sistemas de comunicaciones avanzados, especialmente sobre los basados en 16-QAM. Mediante la aplicación de los conceptos estudiados teóricamente en la asignatura y de las tecnologías explicadas en el laboratorio, se ha conseguido implementar todos los módulos del sistema propuesto, logrando el objetivo principal del proyecto.

Para ello, ha sido necesario un análisis en profundidad de la documentación dada por Xilinx en referencia con el funcionamiento de cada uno de los bloques a desarrollar. Este paso ha sido de gran importancia para garantizar una correcta configuración de los mismos.

Por otro lado, la validación del funcionamiento de los módulos se ha conseguido a partir del análisis cuantitativo y cualitativo de los resultados obtenidos en las simulaciones funcionales. Por ello, se han diseñado ficheros de testbench para cada módulo implementado. La continua evaluación del código ha permitido en ciertos momentos el reajuste de las configuraciones realizadas con el fin de corregir errores y optimizar el sistema.

Finalmente, se puede expresar que se ha logrado una integración completa de todos los módulos de forma exitosa mediante el desarrollo de procesos de lectura y escritura de ficheros de entrada y salida en el testbench para conseguir la cohesión en todo el sistema.