



PROYECTO FINAL

Diseño e implementación de dispositivo emisor de señales codificadas para Sistemas de Posicionamiento Ultrasónico en Entornos Marinos

ALUMNA
Laskowski Sofía Micaela

**Para la obtención del título de
Ingeniería Electrónica**

Directores
Dr. Colombo Martín
Dr. Murano Santiago

**Departamento de
Ingeniería Electrónica**

15 de septiembre de 2023

A mi familia

Agradecimientos

En primer lugar, quiero agradecer a mis padres, quienes me brindaron la posibilidad de estudiar una carrera universitaria y con cariño me impulsaron a perseguir mis metas y alcanzar mis objetivos. Agradezco también a mis hermanos, por el acompañamiento y el apoyo constante en esta etapa. Sin el sostén de mi familia (Chispi, Terry y Blue también), este proyecto no habría sido posible.

Quiero también agradecer a mis directores, el Dr. Santiago Murano y el Dr. Martín Colombo, quienes me guiaron a lo largo del proyecto y me ayudaron a resolver los problemas que fueron surgiendo.

Por último, a los miembros del Grupo de investigación (GIPIS), que siempre estuvieron dispuestos a dar una mano cuando lo necesité.

Resumen

Las comunicaciones acústicas subacuáticas poseen diversas aplicaciones, como la exploración del lecho marino, el monitoreo de ecosistemas acuáticos y particularmente, el posicionamiento de vehículos autónomos subacuáticos (AUVs). Los sistemas de posicionamiento en entornos subacuáticos cuentan con emisores que utilizan esquemas de codificación y de modulación capaces de enfrentar la fenomenología indeseada inherente a los entornos acuáticos. En el marco del Proyecto Final de carrera, se realizó el diseño de un emisor de señales codificadas y moduladas con el potencial de ser utilizado en aplicaciones de comunicaciones subacuáticas. Para la implementación del emisor se utilizó una placa de desarrollo Nexys 4 DDR, configurada por medio del lenguaje de descripción de hardware VHDL. El proyecto consistió en el diseño de un controlador de memoria SD, un controlador de módulo PmodDA2, con su correspondiente interfaz, para la transmisión de las señales codificadas y moduladas almacenadas en una memoria microSD. Los bloques mencionados se diseñaron, simularon e implementaron exitosamente para la obtención de un emisor funcional que se comporta según lo previsto.

Palabras clave: Comunicaciones Acústicas Subacuáticas, Sistemas de Posicionamiento, VHDL, FPGA.

Abstract

Underwater acoustic communications have diverse applications, such as seafloor exploration, monitoring of aquatic ecosystems, and particularly, positioning autonomous underwater vehicles (AUVs). Positioning systems in underwater environments rely on transmitters that use coding and modulation schemes capable of coping with the undesired phenomenology inherent to aquatic environments. Within the framework of the Final Project, a transmitter of coded and modulated signals, with the potential to be used in underwater communications applications, was designed. For the implementation of the transmitter, a Nexys 4 DDR development board was used, configured using VHDL hardware description language. The project consisted in the design of an SD memory controller, a PmodDA2 module controller, with its corresponding interface, for the transmission of coded and modulated signals stored in a microSD memory. The aforementioned blocks were designed, simulated and successfully implemented to obtain a functional transmitter that behaves as expected.

Keywords: Underwater Acoustic Communications, Positioning Systems, VHDL, FPGA.

Índice general

1	Introducción	1
1.1.	Importancia y alcance del problema	1
1.2.	Objetivos y alcance del trabajo	1
1.3.	Organización del informe	2
2	Estado del arte	4
2.1.	Antecedentes del proyecto de investigación: motivación	4
2.2.	Esquemas de codificación y modulación	5
2.2.1.	Esquemas de codificación	5
2.2.2.	Esquemas de modulación	6
2.3.	Placa de Desarrollo Nexys 4 DDR	10
2.3.1.	Puertos Pmod	12
2.3.2.	Ranura para memorias microSD	12
2.3.3.	Lenguaje de descripción de hardware	13
2.3.4.	Software de diseño	14
2.4.	Conceptos básicos de memorias SD	15
2.4.1.	Clasificación de las memorias en función de su capacidad	15
2.4.2.	Clasificación de las memorias en función de su velocidad	16
2.4.3.	Pines de entrada y salida de la memoria	17
2.4.4.	Protocolo de comunicación entre la memoria SD y el host	18
2.5.	Módulo Periférico PmodDA2	23
2.5.1.	Generalidades del módulo	23
2.5.2.	Interfaz con el Pmod	24
2.6.	Reformulación de los objetivos del proyecto	25
3	Lectura de la memoria SD	27
3.1.	Almacenamiento de las señales codificadas y moduladas en la memoria SD	27
3.1.1.	Formato de los datos	27
3.1.2.	Acceso a las distintas señales codificadas y moduladas	28
3.1.3.	Instrucciones para guardar las señales en la memoria	29
3.2.	Controlador de memoria SD	32
3.2.1.	Esquema básico del sistema a diseñar	32
3.2.2.	Diseño del controlador de memoria SD	33
3.2.3.	Simulación del bloque controlador de memoria SD	39
3.2.4.	Implementación del bloque controlador de memoria SD	39
3.3.	Bloque host para la puesta a prueba del controlador	42
3.3.1.	Host para la lectura de 10 bloques consecutivos	42
3.3.2.	Simulación para el host para la lectura de 10 bloques consecutivos	48
3.3.3.	Implementación del host para la lectura de 10 bloques consecutivos	49

4	Controlador del módulo PmodDA2	51
4.1.	Formato de los datos de entrada	51
4.2.	Diseño del controlador del módulo PmodDA2	51
4.3.	Simulación funcional del controlador del PmodDA2	56
4.4.	Implementación del controlador del PmodDA2	57
5	Interfaz entre el controlador y el DAC	60
5.1.	Interfaz directa	60
5.1.1.	Esquema de la interfaz directa entre el controlador de memoria y el Pmod	60
5.1.2.	Simulación de la interfaz directa	61
5.1.3.	Implementación de la interfaz directa	63
5.2.	Interfaz con memoria	63
5.2.1.	Esquema de la interfaz con la memoria RAM auxiliar	63
5.2.2.	Simulación de la interfaz con memoria auxiliar	66
5.2.3.	Implementación de la interfaz con memoria auxiliar	67
5.3.	Incorporaciones finales al diseño	67
5.3.1.	Ajuste de la frecuencia de muestreo	67
5.3.2.	Selección de señal	69
5.3.3.	Disparo por estímulo externo	72
5.4.	Resultados	74
6	Conclusión	77
6.1.	Conclusiones del proyecto	77
6.2.	Trabajo a futuro	78

Índice de figuras

2.1. Generación de la señal modulada con OFDM a partir de una secuencia Zadoff-Chu	10
2.2. Placa de desarrollo Nexys 4 DDR.	11
2.3. Esquema con la vista frontal de uno de los puertos Pmod de la Nexys 4 DDR.	12
2.4. Interfaz entre la Artix-7 y el puerto de la microSD en la Nexys 4. . . .	13
2.5. Símbolos de las distintos tipos de memorias SD en función de su almacenamiento.	16
2.6. Memorias SD de distintas clases de velocidad.	17
2.7. Asignación de pines de tarjeta microSD.	17
2.8. Formato de la respuesta R1.	20
2.9. Formato de la respuesta R7.	20
2.10. Diagrama de flujo de la inicialización de una memoria SD para el modo de operación SPI.	21
2.11. Operación de lectura de un bloque.	22
2.12. Error en la lectura de un bloque.	23
2.13.	23
2.14. Gráfico temporal del DAC121S101 del PmodDA2.	24
2.15. Formato de los datos enviados al PmodDA2 para ser convertidos. . .	25
2.16. Diagrama en bloques simplificado del emisor a ser diseñado.	25
3.1. Esquema básico del controlador de memoria, la memoria SD y el host. .	27
3.2. Eliminación de los retornos de carro en el software Notepad++. . . .	31
3.3. Utilización del editor hexadecimal HxD para la creación del archivo binario de datos.	31
3.4. Esquema del sistema Controlador - Host.	32
3.5. Máquina de estados general del controlador de memoria SD.	33
3.6. Diagrama de flujo simplificado de la rutina de inicialización del controlador de memoria SD.	37
3.7. Funciones TX_AND_RX y SEND_PULSES.	37
3.8. Diagrama de flujo simplificado de la rutina de lectura de un bloque de memoria.	38
3.9. Funciones RX_BITS y SEND_PULSES.	38
3.10. Simulación del controlador de memoria.	39
3.11. Implementación del controlador de memoria en la Nexys 4 DDR para la verificación del controlador.	41
3.12. Diagrama de bloques del controlador y el host diseñados para la prueba del controlador.	43
3.13. Diagrama de flujo del proceso de lectura.	46
3.14. Diagrama de flujo del bloque Multibloque.	47
3.15. Diagrama de flujo del proceso de comparación.	48
3.16. Resultado de prueba de implementación en placa.	50

4.1. Esquema del controlador del PmodDA2.	52
4.2. Proceso Serialización.	53
4.3. Proceso Actualización de salidas.	54
4.4. Proceso Estado de Reserva.	55
4.5. Máquina de estados principal del controlador del PmodDA2.	55
4.6. Simulación del controlador del PmodDA2.	57
4.7. Disposición de los instrumentos utilizados para las pruebas experimentales.	58
4.8. Visualización en el osciloscopio de una onda triangular para la verificación del controlador del PmodDA2.	59
5.1. Esquema de la interfaz directa entre el controlador de memoria SD y el PmodDA2.	61
5.2. Simulación de la interfaz directa entre el controlador de memoria y el PmodDA2.	62
5.3. Captura del osciloscopio con la implementación de la interfaz directa.	63
5.4. Esquema de la interfaz con la memoria auxiliar.	64
5.5. Diagrama de flujo del bloque Administrador de Direcciones.	66
5.6. Simulación del sistema con la interfaz con memoria auxiliar.	67
5.7. Capturas del osciloscopio con la implementación de la interfaz con memoria RAM auxiliar.	68
5.8. Captura del osciloscopio con la señal SYNC que controla la conversión de datos del PmodDA2	69
5.9. Selección de la señal a transmitir por medio de interruptores en la Nexys 4 DDR.	70
5.10. Capturas del osciloscopio de las señales elegidas por medio de los interruptores de la Nexys 4 DDR.	72
5.11. Diagrama en bloques simplificado del emisor.	73
5.12. Captura del osciloscopio con la repetición de las señales cada 1 segundo.	74
5.13. Fragmentos las señales codificadas y moduladas.	76

Índice de tablas

2.1. Asignación de pines de tarjeta microSD.	18
2.2. Formato de los comandos.	19
2.3. Pines de entrada del PmodDA2.	24

1. Introducción

1.1. Importancia y alcance del problema

La transmisión de señales acústicas en medios subacuáticos, cuenta con diversas aplicaciones como: el monitoreo de la vida marina, exploración de hidrocarburos fósiles en el lecho marino, sensado de la contaminación acuática y la exploración submarina por medio de vehículos autónomos subacuáticos (AUVs). Particularmente, en la región costera de la ciudad de Comodoro Rivadavia, ciudad caracterizada por la explotación de petróleo y gas, resulta de gran interés realizar un sensado de distintos indicadores físicos y biológicos del mar. Del mismo modo, en otras localidades donde no se realiza un correcto tratamiento de las aguas residuales, se vuelve fundamental el control de las variables ambientales que garanticen la preservación de los ecosistemas marinos. Por otro lado, contar con la ubicación absoluta o relativa de un nodo, que puede ser un AUV, o un robot subacuático que recopila información en una red, proporciona datos de gran importancia para posteriores aplicaciones o procesamiento de datos por parte de diferentes grupos científicos.

En este sentido, el desarrollo de sistemas de posicionamiento que se adapten a las características de la región, es de gran importancia. Algunas de estas aplicaciones, requieren del despliegue de nodos de referencia sobre la superficie del mar, permitiendo georeferenciar los sensores que se encuentren sumergidos. Para el desarrollo de los sistemas de posicionamiento mencionados se suelen utilizar dispositivos de hardware reconfigurable, a partir de los cuales es necesario transmitir la información almacenada, en forma de señales codificadas y moduladas.

En el Proyecto de Investigación “Modelización y Adaptación de Canales Acústicos en Sistemas Multiportadora Empleados en Redes de Sensores Subacuáticos” del año 2018, en el cual se enmarca el presente proyecto, se estudia la transmisión de señales acústicas en medio subacuáticos.

1.2. Objetivos y alcance del trabajo

En el presente trabajo se realizó la implementación de un sistema autónomo que puede utilizarse como emisor de señales acústicas codificadas y moduladas para estimar la posición del receptor a partir de las medidas de tiempos de vuelo. El módulo diseñado cuenta con la capacidad de almacenar en una memoria SD un conjunto de señales codificadas y moduladas, seleccionar la señal a transmitir y realizar la transmisión. El sistema puede también ser sincronizado con otros emisores para obtener una emisión simultánea de los códigos modulados.

El objetivo global del trabajo consiste en la transmisión de las señales mediante el uso de dispositivos de hardware reconfigurables en baja potencia. Posteriormente, y con la eventual adición de un módulo de adaptación de impedancia y amplificación de potencia, el emisor diseñado puede contar con la capacidad de realizar transmisiones en medios subacuáticos.

Los objetivos parciales del trabajo son:

A. Estudio y comprensión de los esquemas de codificación y modulación a utilizar, basado en los estudios obtenidos en el proyecto de investigación.

Se realizará un análisis de la bibliografía actual relacionada con los temas a abordar, para comprender los conceptos necesarios que permitan el funcionamiento de las secuencias y sus propiedades de correlación. También se estudiarán los parámetros utilizados para la evaluación del desempeño de distintas familias de códigos. Además, se analizarán los esquemas de modulación habituales para adaptar la señal a transmitir al canal de interés, teniendo en cuenta las restricciones en frecuencia y ancho de banda.

B. Diseño e implementación del sistema emisor digital de señales acústicas codificadas.

Se realizará una búsqueda y selección de plataformas hardware que cumplan con los requerimientos eléctricos, de potencia y frecuencia del transductor ultrasónico (emisor). El hardware seleccionado deberá garantizar autonomía y bajo consumo de energía, así como versatilidad de configuración para la realización de pruebas con diferentes esquemas de codificación y modulación. Se realizará el diseño del esquema de emisión, que se encargará de adquirir la señal codificada y modulada, almacenada en una memoria SD conectada al hardware reconfigurable, y a su posterior transmisión por medio de un conversor Digital-Analógico (DAC). El sistema además permitirá al usuario elegir entre una serie de señales prediseñadas, sin la necesidad de reconfigurar el hardware utilizado.

C. Validación del sistema diseñado por medio de simulaciones.

Se realizará a través del armado de bancos de prueba o test benches en el software de simulación integrado en el Vivado.

D. Validación del sistema diseñado por medio de implementaciones.

Se realizará a través de pruebas experimentales reales en la placa de desarrollo, verificando que el diseño se comporta según lo esperado.

1.3. Organización del informe

Para cumplir con los objetivos planteados en la sección anterior, se comenzó por un estudio del conjunto de señales codificadas y moduladas a transmitir, analizando las ventajas y desventajas de cada esquema de codificación y modulación. Posteriormente se realizó el estudio de las características de la capa

física de las memorias SD, desde su distribución de pines hasta su protocolo de inicialización. De la misma manera, se analizaron las hojas de datos y manuales del conversor digital-analógico (DAC) y la placa de desarrollo a ser utilizada. Los principales conceptos de las temáticas mencionadas se presentan en el capítulo 2.

Los capítulos 3, 4 y 5 cuentan con el diseño, la simulación y las implementaciones de distintos módulos del emisor siguiendo la lógica del diseño modular descendente. En la etapa de diseño, se presentan los diagramas en bloques simplificados y los diagramas de flujo¹ de los procesos principales de cada bloque. En la etapa de simulación se presentan los resultados de simulaciones de bancos de prueba, diseñados en el simulador integrado de Vivado. Por último, en las subsecciones de implementación se verifica en la placa de desarrollo la correcta respuesta de los bloques.

En el capítulo 3 se realiza el desarrollo del controlador de memoria SD para la lectura de la memoria SD desde la placa de desarrollo. Posteriormente, en el capítulo 4 se presenta el diseño del controlador del DAC para la conversión y transmisión de muestras digitales. En el capítulo 5 se diseña la interfaz entre los bloques desarrollados en los capítulos 3 y 4. En este capítulo también se incorporan los últimos componentes necesarios para el correcto funcionamiento del emisor. Para finalizar, se presentan los resultados obtenidos a la salida del DAC al transmitir distintas señales codificadas y moduladas.

Por último, en el capítulo 6 se presentan las conclusiones del proyecto junto con propuestas de mejoras para realizar a futuro.

¹A lo largo del proyecto se prioriza la explicación de los diseños por medio de diagramas de flujo, en lugar de presentar las descripciones de hardware, debido a la extensión de las mismas. No obstante, los principales códigos de cada capítulo se pueden encontrar en el Anexo.

2. Estado del arte

Para el desarrollo del proyecto, se abarcaron distintas temáticas vinculadas a las señales a ser transmitidas, así como también las especificaciones del hardware a ser utilizado. En el siguiente capítulo se presentan los principales conceptos estudiados. En primera instancia, se presentan los antecedentes del proyecto de investigación, para luego analizar los esquemas de codificación y modulación para la transmisión de señales, haciendo hincapié en sus propiedades de correlación. Posteriormente se describen las principales características de la placa de desarrollo empleada, la Nexys 4 DDR, como sus puertos y el lenguaje de descripción de hardware empleado. Luego se desarrollan algunos conceptos básicos de las memorias SD, comenzando por sus características físicas y profundizando en el protocolo de comunicación SPI del Bus de datos para el intercambio de información con la FPGA. Finalmente, se describen las características del conversor digital-analógico a ser utilizado, el PmodDA2, y se presenta un diagrama en bloques simplificado con los principales bloques del emisor a ser diseñado.

2.1. Antecedentes del proyecto de investigación: motivación

Conicionados por la rápida atenuación que sufren las señales electromagnéticas a pocos metros de profundidad, los sistemas de comunicaciones y posicionamiento subacuáticos suelen emplear señales acústicas. Como resultado, algunas ondas acústicas pueden llegar a recorrer centenas de kilómetros bajo algunas condiciones. Adicionalmente, al propagarse a menor velocidad que las ondas electromagnéticas, se simplifica su detección en sistemas de localización o sincronización.

Por lo general, los sistemas de posicionamiento en entornos subacuáticos utilizan técnicas de multilateración, que involucran la resolución de sistemas de ecuaciones que tienen en cuenta las relaciones espaciales entre los nodos emisores y el receptor que se desea ubicar [29]. El método más utilizado para medir las relaciones espaciales entre objetos a partir de emisiones acústicas consiste en la medición del tiempo de vuelo (TOF, Time Of Flight) o propagación de las señales emitidas [28]. Dado que la velocidad de propagación en el medio es conocida, se puede determinar la distancia entre objetos, combinando la velocidad de transmisión con el tiempo de propagación. Para ello, es importante utilizar esquemas de codificación con buenas propiedades de correlación, e implementaciones eficientes, que permitan el funcionamiento del sistema en tiempo real.

Existe un gran número de códigos o secuencias que son utilizados en sistemas de posicionamiento en función de sus propiedades y performance en distintos

entornos. Particularmente, en el proyecto de investigación en el que se enmarca el presente proyecto, se ha realizado un estudio de distintos códigos y su desempeño en sistemas de posicionamiento aéreo y subacuático. Entre ellos se destacan los Conjuntos de Secuencias Complementarios (CSS, Complementary Set of Sequences), que cuentan con buenas propiedades de correlación y pueden ser implementados con correladores eficientes en plataformas de arreglos lógicos programables [13][23]. Por otro lado, las secuencias Zadoff-Chu (ZC) [9], además de presentar óptimas propiedades de correlación, tienen un buen comportamiento frente al efecto Doppler. Por último, las secuencias Kasami [17], cuentan con buenas propiedades de correlación cruzada tanto para secuencias periódicas como aperiódicas.

2.2. Esquemas de codificación y modulación

2.2.1. Esquemas de codificación

Para las secuencias que se presentan a continuación, es importante analizar sus propiedades de correlación aperiódica, puesto que esto contribuye a la correcta detección de la señal por parte del emisor.

Secuencias Kasami

Las secuencias Kasami [21] están compuestas por $K_{kas} = 2^{N_{kas}/2}$ secuencias binarias de longitud $L_{kas} = 2^{N_{kas}} - 1$ donde N_{kas} es un número natural par. La expresión de una secuencia Kasami $KAS[n]$ está dada por la ecuación 2.1:

$$KAS_k[n] = m_1[n] \oplus D^l m_2[n] \quad (2.1)$$

donde:

$\oplus$: representa la suma módulo-2.

$m_1[n]$: es una Maximal Length Sequence (MLS) [27] de longitud L_{kas} .

$m_2[n]$: se obtiene al diezmar $m_1[n]$ por un factor $q = 2^{N_{kas}}$ y luego concatenar este resultado q veces.

$D^l m_2[n]$: se obtiene al desplazar cíclicamente l posiciones la secuencia $m_2[n]$.

Las secuencias Kasami cuentan con buenas propiedades de correlación periódica y propiedades de correlación cruzada adecuadas [25].

Secuencias Zadoff-Chu

Las secuencias Zadoff-Chu [12] [9] son códigos polifásicos de amplitud unitaria. Una secuencia Z_{r_k} de longitud L_{ZC} se define a partir de la ecuación 2.2.

$$Z_{r_k}[l] = \begin{cases} \exp j\pi \frac{r_k l^2}{L_{ZC}} & \text{si } L_{ZC} \text{ par} \\ \exp j\pi \frac{r_k l(l+1)}{L_{ZC}} & \text{si } L_{ZC} \text{ impar} \end{cases} \quad (2.2)$$

donde:

$r_k = 1, 2, \dots, L_{ZC} - 1$ entero y co-primo de L_{ZC} .

Para estas secuencias la función de autocorrelación periódica tiene lóbulos laterales nulos (demostrado en [9]), mientras que la función de autocorrelación aperiódica tiene lóbulos laterales pequeños y un SMR (Side to Main Lobe Ratio) máximo adecuado [22]. Asimismo, las secuencias Zadoff-Chu presentan mejores prestaciones frente al efecto Doppler en comparación con las secuencias Kasami [1], al igual que una mayor flexibilidad en cuanto a su longitud.

2.2.2. Esquemas de modulación

Los códigos Kasami y Zadoff-Chu son secuencias que encuentran en banda base, por lo cual, para poder ser transmitidas, y adecuarse a las características en frecuencia del transductor, deben ser moduladas. A continuación se describen los esquemas de modulación empleados, en el proyecto de investigación, para modular las secuencias codificadas.

BPSK - Kasami

En los trabajos desarrollados dentro del grupo de investigación, las secuencias Kasami son moduladas con una modulación BPSK (Binary Phase Shift Keying) [25]. Mediante esta modulación un cambio en el valor de la modulante (binaria) implica un cambio de fase de π radianes de la portadora senoidal.

En una señal BPSK-Kasami, la fase final de la señal a transmitir depende del valor del signo del elemento de la secuencia Kasami. La expresión de la secuencia modulada resultante es:

$$s_{k_{kas}}[n \cdot l] = KAS[l] \cdot \sin \left(2\pi n \frac{f_c}{f_s} \right) \quad (2.3)$$

$$0 \leq n \leq O_f O_c$$

$$0 \leq l \leq L_{kas} - 1$$

donde:

$s_{k_{kas}}[n \cdot l]$: es el código Kasami modulado para el k-ésimo emisor.

$KAS_k[l]$: es un elemento de la secuencia Kasami.

f_s : es la frecuencia de muestreo.

f_c : es la frecuencia de portadora.

O_c : es el número de ciclos de portadora por cada elemento de secuencia.

$O_f = f_s / f_c$: es el factor de sobremuestreo.

PSK - Zadoff-Chu

Las partes reales e imaginarias de las secuencias Zadoff-Chu se modulan con portadoras ortogonales PSK (Phase Shift Keying), obteniéndose la siguiente expresión :

$$C_{f_c}[n] = \cos\left(2\pi n \frac{f_c}{f_s}\right) - j \cdot \sin\left(2\pi n \frac{f_c}{f_s}\right)$$

$$0 \leq n \leq O_f O_c$$

$$0 \leq l \leq L_{ZC} - 1$$

$$s_{k_{QPSK}}[n \cdot l] = [\text{Re}(Z_{r_k}[l]) \cdot \text{Re}(C_{f_c}[n]) - \text{Im}(Z_{r_k}[l]) \cdot \text{Im}(C_{f_c}[n])] \quad (2.4)$$

donde:

$s_{k_{QPSK}}[n \cdot l]$: es un elemento de la secuencia Zadoff-Chu modulada para el k-ésimo emisor.

$C_{f_c}[n]$: es la portadora compleja, centrada a una frecuencia f_c .

O_c : es el número de ciclos de portadora por cada elemento de secuencia.

$O_f = f_s / f_c$: es el factor de sobremuestreo.

OCDM - Zadoff-Chu

La modulación OCDM (Orthogonal Chirp Division Multiplexing) se ha introducido recientemente en el campo de los sistemas ultrasónicos [18] [8], para hacer frente a los problemas en el acceso simultáneo de los códigos chirp tradicionales [20].

En el proyecto de investigación del plan de trabajo se utiliza la propuesta que consiste en la emisión de chirps cuasi-ortogonales. La expresión de de las Chirps complejas está dada por la ecuación 2.5:

$$CH_k(t) = CH_p(t) + CH_q(t) = \text{rect}\left[\frac{t}{T_{c_p}}\right] \exp j 2\pi \left(f_{st_{k_p}} t + \frac{1}{2} \mu_{k_p} t^2\right) +$$

$$\text{rect}\left[\frac{t - T_{c_p}}{T_{c_q}}\right] \exp j 2\pi \left(f_{st_{k_q}} (t - T_{c_p}) + \frac{1}{2} \mu_{k_p} (t - T_{c_p})^2\right) \quad (2.5)$$

donde:

$CH_p(t)$ y $CH_q(t)$: son las componentes en fase y cuadratura de la Chirp Compleja.

f_{s_p} y f_{s_q} : son las frecuencias de inicio de las componentes en fase y cuadratura de la Chirp Compleja.

μ_{k_p} y μ_{k_q} : son las Tasas de variación de frecuencia de las componentes en fase y cuadratura de la Chirp.

T_{c_p} y T_{c_q} : son las duraciones de las componentes en fase y cuadratura de la Chirp. $T_{c_p} + T_{c_q} = T_c$ donde T_c es la duración de un período de la Chirp.

A partir de la señal Chirp de valores complejos como símbolo de modulación de códigos Zadoff-Chu, se puede obtener una señal real (similar al caso PSK-Zadoff-Chu). En la ecuación 2.6 se presenta la expresión de la OCDM-Zadoff-Chu:

$$s_{k_{CH}}[n \cdot l] = [\text{Re}(Z_{r_k}[l]) \cdot \text{Re}(CH_k[n]) - \text{Im}(Z_{r_k}[l]) \cdot \text{Im}(CH_k[n])] \quad (2.6)$$

$$0 \leq n \leq N_{T_c}$$

$$0 \leq l \leq L_{ZC} - 1$$

donde:

$CH_k[n]$: es la Chirp compleja.

$s_{k_{CH}}[n \cdot l]$: es la secuencia Zadoff-Chu modulada.

$Z_{r_k}[l]$: es un elemento de la secuencia Zadoff-Chu.

N_{T_c} : es la duración del símbolo Chirp en muestras.

L_{ZC} : es la longitud de la secuencia Zadoff-Chu.

OFDM - Zadoff-Chu

La modulación OFDM (Orthogonal Frequency Division Multiplexing) es una modulación multiportadora en la cual los datos son transmitidos a través de un conjunto de portadoras ortogonales. Los datos son distribuidos en N_{FFT} canales de frecuencia equiespaciados, en un ancho de banda de 0 a $f_s/2$, donde f_s es la frecuencia de muestreo.

Para la implementación del esquema de modulación se utiliza la transformada inversa de Fourier (IFFT). Utilizando el doble de muestras ($N_{FFT} = 2N$) se puede emitir una señal real (propiedad de simetría Hermítica ¹).

La modulación puede ser expresada como:

¹Su conjugado es igual a la función inicial pero con el signo cambiado.

$$s_{k_{OFDM}}[n \cdot l] = \sum_{l=0}^{2N_{OFDM}-1} S_k[l] \cdot e^{j2\pi nl/2N_{OFDM}} \quad (2.7)$$

$$n = 0, \dots, 2N_{OFDM} - 1$$

donde :

$S_k[2N_{OFDM} - l] = S_k[l]^*$ y $S_k[l]^*$ es el complejo conjugado de $S_k[l]$.

$s_{k_{OFDM}}[n]$: es la secuencia Zadoff-Chu modulada.

$S_k[l]$: es símbolo transmitido en la sub-portadora l .

Para ubicar la secuencia Zadoff-Chu dentro de la banda de respuesta apropiada del transductor, cada elemento de la secuencia debe ser colocado en los canales de la IFFT que se correspondan con las frecuencias de salida deseadas. Las canales restantes se completan con ceros. La cantidad de ceros previos Z_L y posteriores Z_H a la secuencia está dada por las expresiones en 2.8.

$$Z_H = \frac{L_{ZC}(f_s - 2f_h)}{2(f_h - f_l)} \quad (2.8)$$

$$Z_L = \frac{2f_l(L_{ZC} + Z_H)}{f_s - 2f_l}$$

La cantidad de ceros depende de la frecuencia de muestreo f_s , el ancho de banda disponible $B = f_h - f_l$ y la longitud de la secuencia L_{ZC} .

Antes de la aplicación de la simetría Hermítica, la secuencia de entrada $S_k[l]$, consiste en un vector de longitud N_{OFDM} , como el que se presenta en la ecuación 2.9.

$$S_k[l] = [0, \text{zeros}(1 : Z_L), Z_{r_k}[0], \dots, Z_{r_k}[L_{ZC} - 1], \text{zeros}(1 : Z_H)] \quad (2.9)$$

En la Figura 2.1 se presenta la generación de la señal codificada y modulada OFDM-Zadoff-Chu, con los correspondientes canales de la IFFT alimentados con ceros.

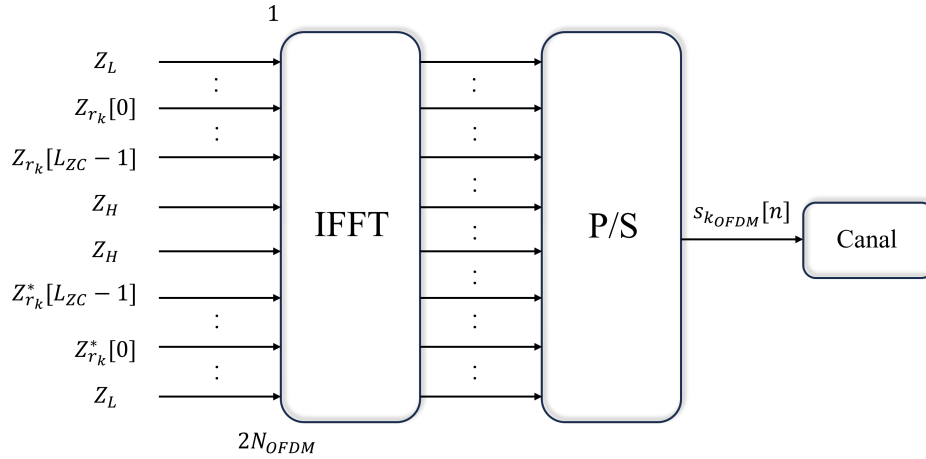


Figura 2.1: Generación de la señal modulada con OFDM a partir de una secuencia Zadoff-Chu

Fuente: Murano [25].

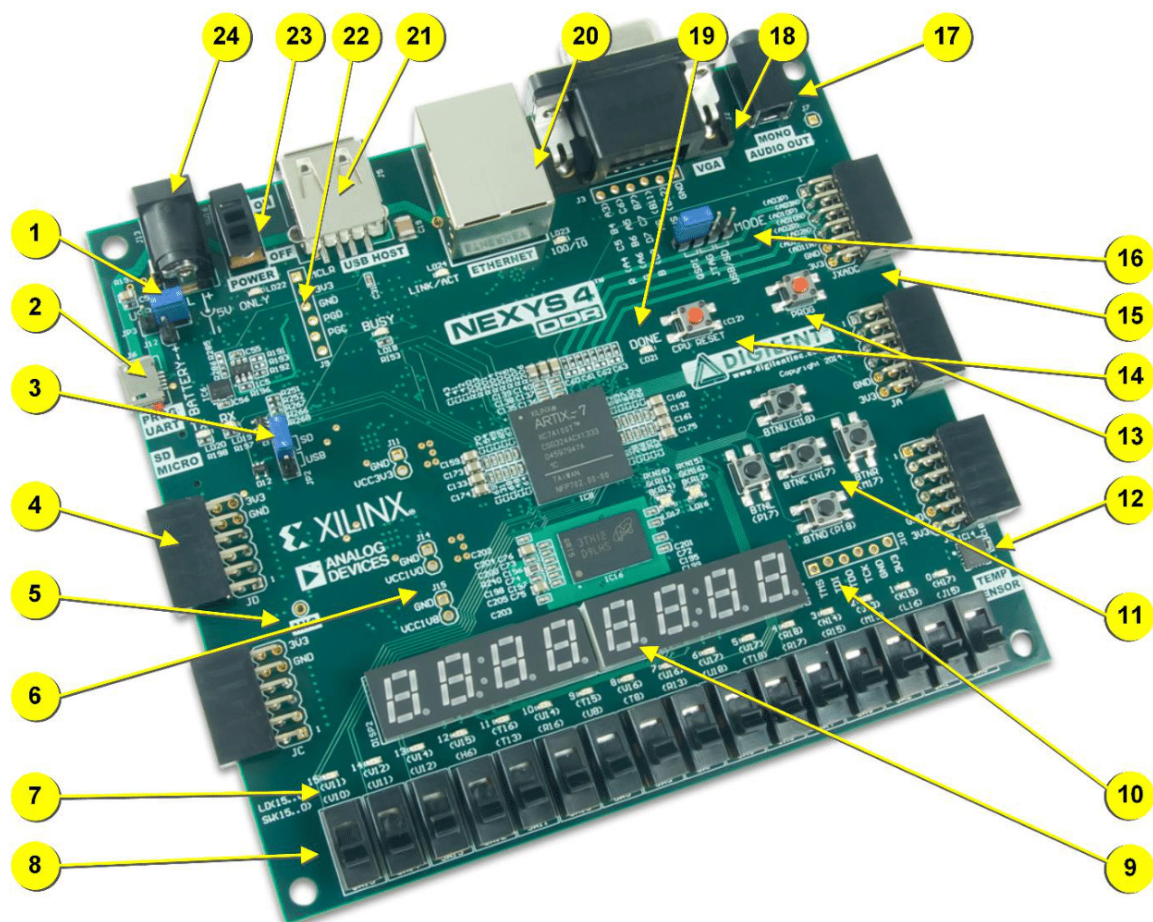
2.3. Placa de Desarrollo Nexys 4 DDR

La Nexys 4 DDR es una placa de desarrollo de Digilent que cuenta con un circuito digital completo y listo para ser utilizado. Está compuesta por una FPGA (Field Programmable Gate Array) Artix-7 de Xilinx. Un dispositivo FPGA es un circuito digital integrado VLSI (Very Large-Scale Integration) que puede ser configurado por el usuario. Está compuesta por un arreglo de una gran cantidad de bloques lógicos, rodeados por bloques reconfigurables de entrada y salida, y conectado por medio de interconexiones programables. Un bloque lógico de la FPGA está compuesto por LUTs (Look Up Tables), multiplexores, compuertas y flip-flops.

Las características principales de la FPGA Artix-7 son:

- 15,850 bloques lógicos, cada uno con 4 LUTs de 6 entradas y 8 flip-flops.
- 4,860 Kbits de BRAM (Block RAM).
- Seis bloques de gestión de reloj, cada uno con PLL (phase-locked loop).
- 240 segmentos DSP (Digital Signal Processing).
- Velocidades internas de reloj que superan los 450 MHz.
- Conversor analógico-digital en chip (XADC) .

En conjunto con su FPGA de alta capacidad (XC7A100T-1CSG324C), la Nexys 4 DDR incluye memorias externas, puertos USB, Ethernet y lector de memorias microSD. Por lo tanto, esta placa puede ser utilizada para un gran número de aplicaciones, desde diseños básicos de introducción a los circuitos digitales, hasta circuitos de mayor complejidad como procesadores embebidos. En la Figura 2.2(a) se presenta la placa Nexys 4 DDR con la indicación de sus puertos y periféricos.



(a)

Callout	Component Description	Callout	Component Description
1	Power select jumper and battery header	13	FPGA configuration reset button
2	Shared UART/ JTAG USB port	14	CPU reset button (for soft cores)
3	External configuration jumper (SD / USB)	15	Analog signal Pmod port (XADC)
4	Pmod port(s)	16	Programming mode jumper
5	Microphone	17	Audio connector
6	Power supply test point(s)	18	VGA connector
7	LEDs (16)	19	FPGA programming done LED
8	Slide switches	20	Ethernet connector
9	Eight digit 7-seg display	21	USB host connector
10	JTAG port for (optional) external cable	22	PIC24 programming port (factory use)
11	Five pushbuttons	23	Power switch
12	Temperature sensor	24	Power jack

(b)

Figura 2.2: Placa de desarrollo Nexys 4 DDR.
Fuente : Nexys4 DDR™ FPGA Board Reference Manual [10].

De los puertos y periféricos presentes, se pueden destacar los siguientes para la

aplicación del proyecto:

- 16 switches de usuario.
- 16 LEDs de usuario.
- 2 LEDs Tricolor.
- Puerto USB-JTAG para programación y comunicación con la FPGA.
- Conector para tarjeta Micro SD.
- Cuatro puertos Pmod.

2.3.1. Puertos Pmod

Los puertos Pmod (acrónimo de Peripheral Module) están compuestos por 2x6 conectores hembra. Cada puerto Pmod de 12 pines cuenta con dos pines de alimentación VCC de 3.3 V (pines 6 y 12), dos pines de GND (pines 5 y 11) y 8 pines de señales lógicas. En la Figura 2.3 se presenta un esquema con la vista frontal de un puerto Pmod.

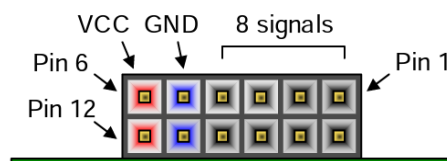


Figura 2.3: Esquema con la vista frontal de uno de los puertos Pmod de la Nexys 4 DDR.

Fuente : Nexys4 DDR™ FPGA Board Reference Manual [10].

Digilent cuenta con una gran colección de accesorios Pmod que pueden ser conectados a los puertos Pmod para ampliar la capacidad de la placa de desarrollo, agregando nuevas características y funcionalidades. Existen módulos ADC (conversor analógico-digital), DAC (conversor digital-analógico), módulos LCD, GPS, drivers para motores, entre otros².

2.3.2. Ranura para memorias microSD

La Nexys 4 DDR cuenta con una ranura microSD, que puede ser utilizada tanto para la configuración de la FPGA así como para aplicaciones del usuario. No obstante, para el acceso de la memoria por parte del usuario, se debe implementar un host controlador de memoria SD. Este será el encargado de comandar la memoria para cumplir con las funcionalidades requeridas para la aplicación particular del usuario.

En la Figura 2.4 se presenta Interfaz entre la Artix-7 y el puerto de la microSD en la Nexys 4.

²<https://digilent.com/reference/pmod/start>

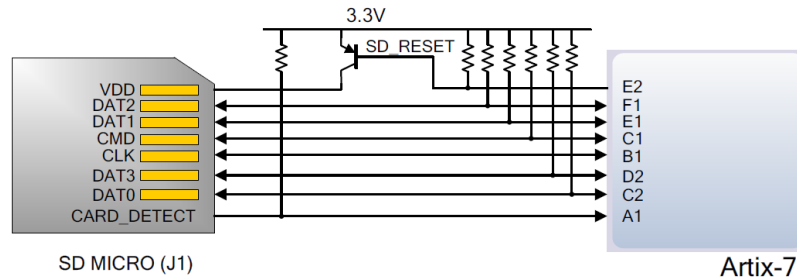


Figura 2.4: Interfaz entre la Artix-7 y el puerto de la microSD en la Nexys 4.
Fuente : Nexys4 DDR™ FPGA Board Reference Manual [10].

2.3.3. Lenguaje de descripción de hardware

El diseño de circuitos digitales en FPGA requiere de algún lenguaje de descripción de hardware HDL (Hardware Description Language) como ABEL, VHDL o Verilog. El lenguaje utilizado para el proyecto fue el VHDL, cuyas siglas provienen de HDL y VHSIC (Very High Speed Integrated Circuits). Para que un circuito diseñado en VHDL pueda ser sintetizado debe contar con tres unidades de diseño: librerías, entidad y arquitectura.

Las *librerías* representan un conjunto de extractos de código VHDL que pueden ser reutilizados o compartidos entre diseños. Los extractos de código de una librería pueden presentarse en forma de componentes, funciones o procedimientos agrupados en paquetes. En el Código 2.1 se presenta la declaración de una librería y los paquetes a ser utilizados para el diseño.

```

1 library library_name;
2 use library_name.library_package.package_part;

```

Código 2.1: Declaración de las librerías en VHDL.

En la *entidad* se especifican las entradas y salidas del circuito diseñado. Comienza por el identificador de la entidad como se muestra en el Código 2.2. Los pines del circuito se definen por medio de **port** y a su vez se debe definir el tamaño y la dirección del puerto.

```

1 entity entity_name is
2 port ( port_name : signal_mode signal_type;
3       port_name : signal_mode signal_type;
4       ... );
5 end entity_name;

```

Código 2.2: Declaración de la entidad en VHDL.

La *arquitectura* se asocia a una entidad y describe el funcionamiento del circuito diseñado. La sintaxis de una arquitectura se presenta en el Código 2.3.

```
1 architecture architecture_name of entity_name is
2 [declarations]
3 begin
4 (code)
5 end architecture_name;
```

Código 2.3: Declaración de la arquitectura en VHDL.

En la sección de declaración de la arquitectura (declarations) se definen las señales internas. Por otro lado, en la sección de código (code) se definen las sentencias concurrentes y los procesos. La arquitectura en sí puede ser interpretada como un conjunto de partes interconectadas de un circuito que se ejecutan en forma concurrente. A diferencia de la programación de un microcontrolador, donde las sentencias se ejecutan siguiendo el orden secuencial en el que son definidas, en VHDL, las sentencias de la arquitectura son concurrentes, incluso en los procesos, que empaquetan un conjunto de sentencias secuenciales, aunque luego se interprete como un circuito concurrente.

2.3.4. Software de diseño

Para el diseño de cualquier circuito digital se deben cumplir los siguientes pasos:

1. **Síntesis** : Se transforma la descripción del comportamiento del circuito en una descripción del circuito a nivel estructural.
2. **Diseño Físico** : Consiste en el proceso de refinamiento o depuración entre las representaciones estructurales y físicas, así como también el análisis de las características eléctricas del circuito.
3. **Verificación** : Se utiliza para determinar si el diseño cumple con las funcionalidades y especificaciones deseadas. Las formas concretas de verificación para el diseño de circuitos digitales son las simulaciones, los análisis temporales y la emulación de hardware.
4. **Puesta a Prueba** : Consiste en la verificación en placa del funcionamiento del circuito.

Para completar estos pasos se utiliza un software de diseño por computadora. En particular, la Nexys 4 DDR es compatible con el software Vivado® Design Suite. Xilinx ofrece versiones libres de las herramientas de diseño, bajo el nombre WebPACK™ por lo cual los diseños pueden ser implementados sin costos adicionales.

Dentro del Vivado se crean los distintos proyectos VHDL a ser presentados en los capítulos de desarrollo. Un proyecto HDL consiste en un conjunto de componentes diseñados, sus respectivos test benches y un archivo *.xdc* (Xilinx Design Constraint) que contiene las direcciones de los puertos físicos de la FPGA y

permite realizar las conexiones entre los pines externos de la FPGA, con los del componente de mayor jerarquía.

2.4. Conceptos básicos de memorias SD

Las tarjetas SD (Secure Digital) son memorias diseñadas para cumplir con los requisitos de desempeño, seguridad y capacidad de los dispositivos portátiles emergentes, como cámaras digitales (fotográficas o de video), celulares, computadoras, entre otros. El estándar para memorias SD se presentó en 1999 como una modernización de las tarjetas MMC (MultiMedia Card) por distintas compañías, dentro de las cuales se destacan Toshiba (ahora KIOXIA), SanDisk y Panasonic. En la actualidad el estándar SD es utilizado por una gran cantidad de fabricantes de memorias y es mantenido por la SD Association ³, a la cual hoy pertenecen más de 800 compañías. Gracias a la existencia de los estándares SD, puede existir una compatibilidad entre dispositivos, lo cual resulta de gran conveniencia para el consumidor.

La SD Association cuenta con una serie de documentaciones disponible en su página web oficial, donde se incluyen las especificaciones del estándar SD. Esta documentación se encuentra dividida en partes:

- Parte 1: Especificaciones de la Capa Física.
- Parte 2: Especificaciones del Sistema de Archivos.
- Parte 3: Especificaciones de la Seguridad.
- Parte 4: Especificaciones de aplicaciones específicas (audio, video, entre otras).

Para el diseño del controlador de memoria SD requerido por la placa de desarrollo Nexys 4 DDR se hace hincapié en la Parte 1 de la especificaciones. La descripción de la capa física incluye tanto la interfaz física de las memorias SD así como su protocolo de comunicación. En las secciones que siguen se presentan algunos de los conceptos principales presentados en la Parte 1 de la documentación de la SD Association, titulada “Physical Layer Simplified Specification” [14] [15].

2.4.1. Clasificación de las memorias en función de su capacidad

Las memorias SD se clasifican según su capacidad en 4 categorías:

- Standard Capacity Memory Cards (SDSC): hasta 2 GBytes de capacidad.
- High Capacity Memory Cards (SDHC): hasta 32 GBytes de capacidad.
- Extended Capacity Memory Cards (SDXC): hasta 2 TBytes de capacidad.
- Ultra Capacity Memory Cards (SDUC): hasta 128 TBytes de capacidad.

³<https://www.sdcard.org>

En la Figura 2.5 se presentan los símbolos de las distintas clases de memorias en función de su capacidad de almacenamiento.



Figura 2.5: Símbolos de las distintos tipos de memorias SD en función de su almacenamiento.

Fuente: SD Capacity [6].

Es importante destacar que los Hosts que sean capaces de acceder a una determinada capacidad de memoria, realizando operaciones de lectura y escritura, podrán también acceder a memorias de menores capacidades. Por ejemplo, un host que puede acceder a memorias SDXC puede también leer o escribir memorias SDHC y SDSC. Sin embargo, a la inversa no es posible, es decir, un host no puede acceder a memorias de mayor capacidad que la de diseño del host.

En el presente proyecto se hace uso de memorias SDHC, puesto que con 32 GBytes de almacenamiento se tiene espacio de sobra para almacenar las señales necesarias. No se utilizan memorias de capacidad standard dada su escasa disponibilidad en el mercado actual.

2.4.2. Clasificación de las memorias en función de su velocidad

Otra forma de clasificar las memorias, es en función de la mínima velocidad de escritura. En la Figura 2.6 se presentan 3 tipos de clases de velocidad y sus respectivas subclasificaciones. Las clases se indican en MB/s, es decir que una memoria SD clase 6 tiene una velocidad mínima de 6 MB/s ($6 \cdot 10^6$ Bytes por segundo). Cabe destacar que la velocidad indicada es la mínima velocidad de escritura y no su velocidad de lectura, por lo que no interfiere con el diseño del controlador de memoria.

Minimum Sequential Write Speed	Speed Class		
	Speed Class	UHS Speed Class	Video Speed Class
Card Image			
90MB/sec			V90
60MB/sec			V60
30MB/sec		U3	V30
10MB/sec	10	U1	V10
6MB/sec	6		V6
4MB/sec	4		
2MB/sec	2		

Figura 2.6: Memorias SD de distintas clases de velocidad.
Fuente : SD Speed Class [7].

2.4.3. Pines de entrada y salida de la memoria

Las memorias SD cuentan con nueve pines cuya enumeración se presenta en la Figura 2.7. Estos pines cuentan con distintas denominaciones en función del modo de operación (modo SD o modo SPI) de la memoria. En la Tabla 2.1 se presentan las denominaciones y descripciones de los pines para el modo SPI.

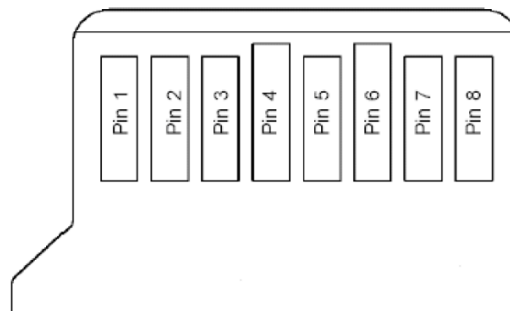


Figura 2.7: Asignación de pines de tarjeta microSD.
Fuente: microSDHC memory card Flash Storage Media [19].

Se observa que para la operación en el modo SPI, hay dos pines que no se utilizan. Por otro lado, los pines de alimentación ya están conectados en la Nexys 4 DDR (Figura 2.4). Por lo tanto, las líneas a conectar al controlador son **SCLK**, **CS**, **DI** y **DO**.

Pines	Name SD	Name SPI	IO Type	Description
1	DAT2	RSV		
2	CD / DAT3	CS	I	Chip Selection (neg true)
3	CMD	DI	I	Data In
4	VDD	VDD	S	Supply Voltage
5	CLK	SCLK	I	Clock
6	VSS	VSS	S	Supply Volyage Ground
7	DAT0	DO	O/PP	Data Out
8	DAT1	RSV	-	Reserved (*)

Tabla 2.1: Asignación de pines de tarjeta microSD.
Fuente: microSDHC memory card Flash Storage Media [19].

Si bien la Nexys 4 DDR es compatible con las memorias SD, cabe destacar que la tensión de alimentación de las memorias SD debe estar comprendida entre 2,7 y 3,6 V (la Nexys 4 DDR alimenta a las memorias SD con 3.3V). Adicionalmente, la frecuencia de reloj máxima durante la inicialización es de 400 kHz, mientras que en operación normal es de 25 MHz.

2.4.4. Protocolo de comunicación entre la memoria SD y el host

La comunicación a través del bus de la memoria SD consiste en flujos de información delimitados por bits de inicio y fin. Estos se pueden clasificar en:

- Comandos: se utilizan para comenzar una operación determinada y son enviados por el host en forma serial.
- Respuestas: son el resultado del comando enviado por el host y se transfieren serialmente desde la memoria SD.
- Datos: se intercambian bidireccionalmente entre el host y la SD.

Existen dos protocolos a través de los cuales un host se puede comunicar con una memoria SD: el protocolo de Bus SD y el protocolo de Bus de SPI (Serial Peripheral Interface). Los datos de la memoria pueden ser leídos o escritos por medio de cualquiera de ellos. El modo SD del bus permite la transferencia de datos a través de un puerto de 4 bits, mientras que el modo SPI permite la transferencia por medio de 1 única línea de bits a través de dos canales (Data In y Data Out). La desventaja principal del protocolo SPI frente al protocolo de bus SD es la pérdida de desempeño, proveniente de la reducción en la cantidad de bits de salida. No obstante, se obtiene como ventaja la disminución en la complejidad del diseño del controlador de memoria (Página 93 [14]).

Una vez que la memoria haya finalizado la rutina de inicialización para uno de los dos protocolos mencionados, los procesos de lectura y escritura se realizarán respetando dicho protocolo. Por lo tanto, la memoria SD mantendrá el modo de operación escogido hasta que sea reiniciada.

El protocolo escogido para la comunicación entre la FPGA y la memoria SD es

el SPI, dado que la cantidad de datos a leer de la memoria no es lo suficientemente significativa como para justificar la utilización del protocolo SD, de mayor complejidad.

Protocolo SPI

El protocolo SPI está orientado a bytes. Se intercambian datos, comandos y respuestas. Cada comando o bloque de datos está compuesto por bytes y está sincronizado con la señal **CS** (Selección de Chip o Chip Select).

A diferencia del protocolo de bus SD, en el protocolo SPI, la memoria SD siempre devuelve una respuesta al recibir un comando, inclusive al producirse un error. En el caso de memorias SDSC los bloques son de tamaño variable y pueden configurarse entre 0 y 512 bytes. Sin embargo, tanto en memorias SDHC (utilizadas en este proyecto) como en memorias SDXC, los bloques son de longitud fija (512 bytes). Como las operaciones de lectura en las memorias SD se realizan por bloques o sectores, al solicitarse una lectura de una memoria SDHC, se leen 512 bytes.

Comandos

Los comandos utilizados para la comunicación entre el host y la memoria SD tienen una longitud de 6 bytes, y su transmisión comienza siempre por el bit más significativo (el que se encuentra más a la izquierda). Todos los comandos están protegidos por un CRC (Cyclic Redundancy Check), para evitar cambios accidentales en los datos. En la Tabla 2.2 se presenta una descripción de la función de los distintos bits del comando de 6 bytes.

Bit position	47	46	[45:40]	[39:8]	[7:1]	0
Width (bits)	1	1	6	32	7	1
Value	'0'	'1'	x	x	x	'1'
Description	Start bit	Transmission bit	Command index	Argument	CRC7	End bit

Tabla 2.2: Formato de los comandos.

Fuente: Physical Layer Simplified Specification [14].

Respuestas

En el modo SPI existen distintos Tokens de Respuesta que son enviados por la SD luego de recibir algún comando por parte del Host. Para el proyecto actual, se pueden destacar los siguientes formatos de posibles respuestas:

- Formato R1: Este token de respuesta se envía luego de cada comando, comenzado por su MSB, que siempre es 0. Los 7 bits restantes son indicadores de error, con el error representado por medio de un 1.

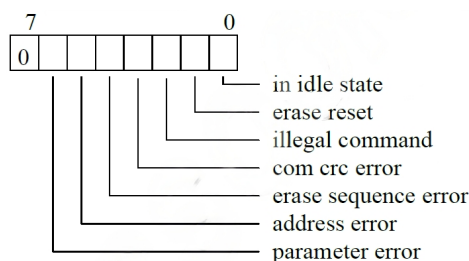


Figura 2.8: Formato de la respuesta R1.
Fuente: Physical Layer Simplified Specification [14].

- Formato R7: Este token de respuesta tiene una longitud de 5 bytes y se envía luego de un comando CMD8. El formato del primer byte es igual al del token R1. Los 4 bytes restantes contienen información del voltaje de operación.

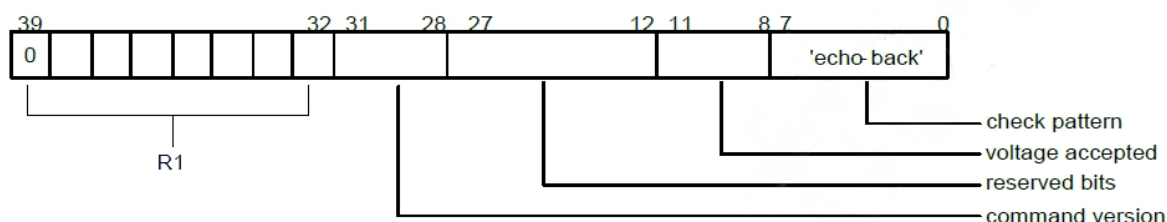


Figura 2.9: Formato de la respuesta R7.
Fuente: Physical Layer Simplified Specification [14].

Inicialización de la tarjeta SD en el modo SPI

Para comenzar con la inicialización de la memoria SD se debe primero alimentarla con la tensión de alimentación mencionada anteriormente y enviar un mínimo de 74 ciclos de clock con CS en alto. Luego de esto se puede dar inicio a la secuencia de inicialización. En la Figura 2.10 se presenta un diagrama de flujo con el intercambio de comandos y sus respectivas respuestas entre la memoria SD y el host durante la inicialización del protocolo SPI.

Se observa que para la correcta inicialización de la SD, el host debe enviar una serie de comandos en un orden determinado⁴. Hay comandos que son obligatorios como el CMD0 y el CMD8, pero hay otros que no, como el CMD58. El objetivo de la rutina de inicialización es verificar que la memoria SD responde correctamente a distintos comandos y que por consiguiente puede comunicarse con el Host.

Para comenzar con la inicialización, el host le envía el comando CMD0 con el pin CS en bajo a la memoria SD. La abreviación de este comando es GO.IDLE.STATE y su función es la de resetear la memoria. Al enviar este comando la memoria SD muestrea la entrada CS y en caso de que se encuentre en un valor bajo, la memoria ingresa al modo SPI y responde con R1 con el bit In Idle State seteado (h01). Si

⁴La lista de comandos completa con su respectiva descripción se encuentra en la Physical Layer Simplified Specification [14], páginas 49-55.

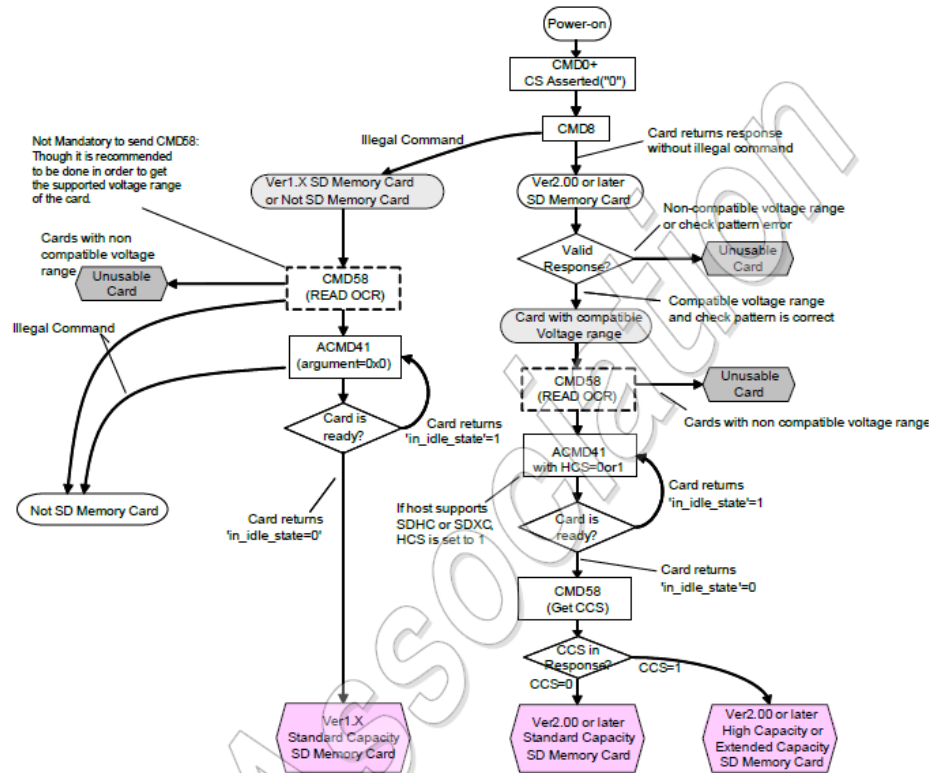


Figura 2.10: Diagrama de flujo de la inicialización de una memoria SD para el modo de operación SPI.

Fuente: Physical Layer Simplified Specification [14]

se verifica esta condición, el host puede enviar el siguiente comando, en el caso contrario vuelve a enviar el CMD0.

El CMD8 se abrevia como SEND_IF_COND y se utiliza para enviar a la memoria información de la tensión de alimentación, a la vez que se le consulta si puede operar en dicho rango de voltaje. El argumento del CMD8 es h000001AA e indica que la memoria puede trabajar en un rango de tensiones de 2.7 a 3.6 V. Como resultado de este comando la memoria SD responde con R7 (R1(h01) + un valor de 32 bits). En este caso el host no debe verificar la respuesta, esta ya le permite avanzar al siguiente comando, dado que la Nexys 4 DDR es compatible con la memoria SD en términos de tensión de alimentación.

Tanto CMD0 como CMD8 deben ser enviados como comandos nativos por lo cual deben contar con un CRC válido. Una vez que la memoria entra en el modo SPI el CRC se deshabilita, por lo cual la memoria no verifica ni el CRC de los comandos, ni el de los datos transferidos.

El siguiente comando a ser enviado es el CMD55 que es el preámbulo del ACMD41. El CMD55 tiene como abreviación APP_CMD y le indica a la memoria que el siguiente comando que recibirá es un comando específico de aplicación y no un comando regular. Posteriormente se envía el CMD41 cuyo argumento es h40000000. El ACMD41 (CMD55 seguido de CMD41) tiene como abreviación SD_SEND_OP_COND. Este comando reservado tiene como función activar el proceso de inicialización en la memoria y dejarla lista para recibir comandos de

lectura y escritura. Como resultado, la memoria tendría que salir del estado inactivo y pasar a un estado activo, es decir bajar el flag “In Idle State” (R1 h00).

El comando que sigue a continuación en el diagrama de flujo de la Figura 2.10 es el comando CMD58 READ_OCR que lee el registro OCR de la memoria. A partir de la respuesta de este comando, se puede evaluar el valor de CCS para así saber si la memoria es SCSD o superior (SDHC o SDXC). Dado que en la actualidad las tarjetas SDSC no se comercializan tanto como otros modelos, se optó por omitir este comando en la rutina de inicialización y limitar el diseño a tarjetas SDHC.

Códigos de Redundancia Cíclica

Al realizar la inicialización de la memoria SD en el modo SPI, el CRC se encuentra desactivado por defecto. No obstante, el comando CMD0 utilizado en la inicialización pertenece al repertorio de comandos del protocolo de bus en modo SD por lo cual debe contener el CRC correcto (h95). Una vez que la memoria se ha configurado en el modo SPI la revisión del CRC para todos los comandos depende del valor seteado a través del comando CMD59 (se puede habilitar el CRC). Otro comando con CRC obligatorio es el CMD8, y el valor correcto es h87. Un error en el CRC se reporta a través del token de respuesta R1.

Operación de lectura en el modo SPI

El protocolo de bus SPI puede realizar tanto operaciones de lectura de un solo bloque o múltiples bloques. Al recibir un comando de solicitud de lectura la memoria SD envía un token de respuesta seguido de un token de datos tal como se observa en la Figura 2.11. El formato de un token de datos está compuesto por un primer byte hFF, seguido de los 512 bytes de datos y 2 bytes de CRC.

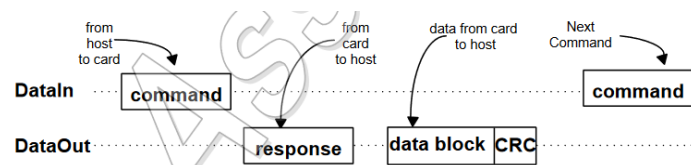


Figura 2.11: Operación de lectura de un bloque.
Fuente: Physical Layer Simplified Specification [14].

Si el bloque de datos es válido, le sucede una palabra de 16 bits con el CRC-CCITT generado por el polinomio estándar $x^{16} + x^{12} + x^5 + 1$.

La dirección de inicio de lectura puede ser cualquiera de las direcciones del rango válido de la memoria SD. En caso de producirse un error en la obtención de los datos, la memoria no transmite ningún dato. En lugar de datos, se envía un token de error al host. En la Figura 2.12 se presenta el caso en el que un proceso de lectura se interrumpe con un token de error en lugar de un bloque de datos.

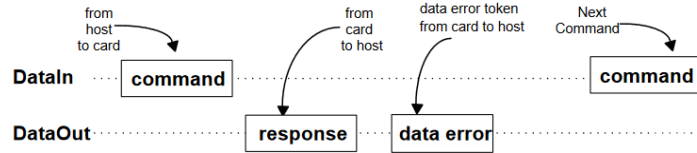


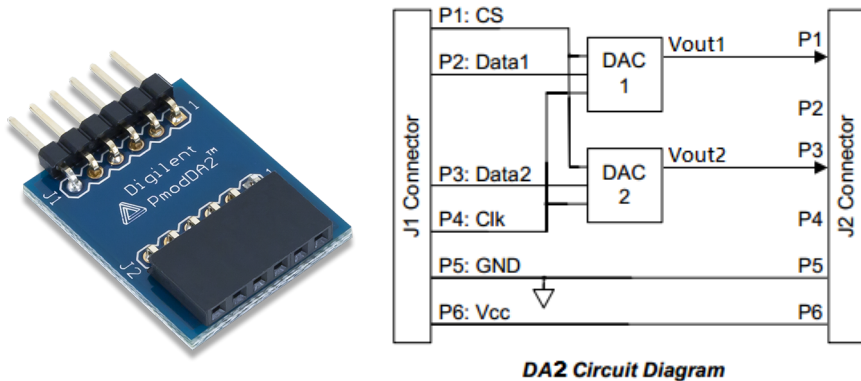
Figura 2.12: Error en la lectura de un bloque.
Fuente: Physical Layer Simplified Specification [14].

2.5. Módulo Periférico PmodDA2

2.5.1. Generalidades del módulo

El PmodDA2 (Figura 2.13(a)) es un módulo periférico (Pmod) conversor analógico-digital (DA) de 12 bits de Digilent, compuesto del DAC121S101 de Texas Instruments. Cuenta con dos canales de conversión por lo cual se pueden realizar conversiones simultáneas de dos entradas digitales en salidas analógicas.

El módulo posee 6 pines GPIO (General-Purpose Input/Output) que se conectan con alguna de los puertos de entrada/salida Pmod de la placa de desarrollo, y 4 pines que contienen las salidas analógicas. El diagrama en bloques simplificado del PmodDA2 se presenta en la Figura 2.13(b). Allí se pueden observar los pines de entrada mencionados, cuya denominación, descripción y valores máximos se presentan en la Tabla 2.3.



(a) Módulo PmodDA2.

(b) Diagrama de bloques del PmodDA2.

Figura 2.13

Fuente: PmodDA2™ Reference Manual [11].

Los pines de salida del PmodDA2 son 4:

- VOUT1 : Salida analógica 1 (P1 en la Figura 2.13).
- VOUT2 : Salida analógica 2 (P3 en la Figura 2.13).
- VCC : Alimentación positiva.
- GND : Ground.

Pin	Señal	Descripción	Valor máximo
1	SYNC	Chip Select	VCC + 0.1 V
2	DATA1	Entrada de datos para el canal 1	VCC + 0.1 V
3	DATA2	Entrada de datos para el canal 1	VCC + 0.1 V
4	SCLK	Clock	30 MHz
5	GND	Ground	-
6	VCC	Alimentación positiva	5.5 V

Tabla 2.3: Pines de entrada del PmodDA2.
Fuente: PmodDA2™ Reference Manual [11].

2.5.2. Interfaz con el Pmod

El PmodDA2 se comunica con la lógica del Host por medio de un protocolo similar al SPI. Para enviar los bits de datos a través de pulsos de reloj (**SCLK**), se debe antes colocar la línea de selección de Chip (**SYNC**) a un valor bajo. Los datos se envían comenzando por el bit más significativo (MSB) y terminando con el menos significativo (LSB). En la Figura 2.14 se presenta un gráfico temporal donde se pueden observar los cambios de la señal **SYNC** para el correcto envío de la palabra a convertir.

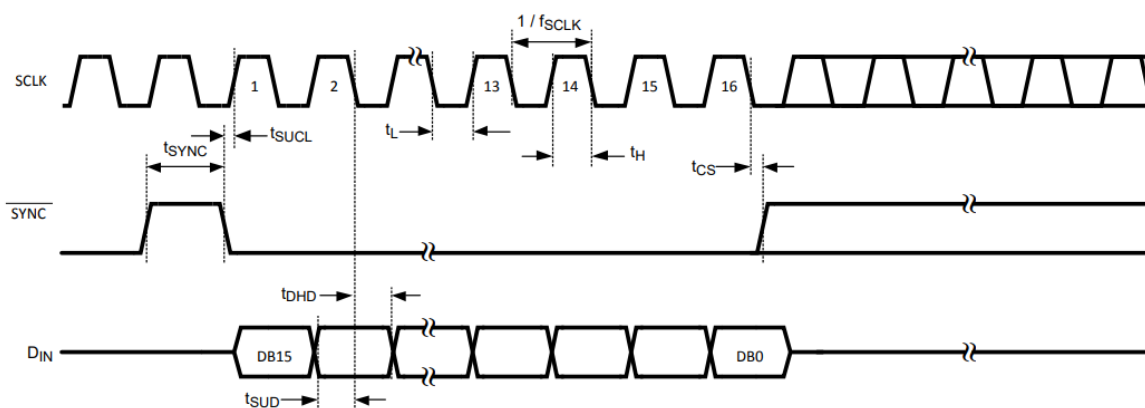


Figura 2.14: Gráfico temporal del DAC121S101 del PmodDA2.
Fuente: DAC121S101 Manual [16].

Previo a los 12 bits de datos a convertir, se deben enviar 4 bits adicionales, de los cuales 2 son Don't care (no importa su valor), y los dos restantes varían en función del modo de funcionamiento. En la Figura 2.15 se presentan los 16 bits a enviar por cada palabra que se desea convertir. Se observa que para la conversión de palabras digitales en operación normal, los bits **PD1** y **PD0** deben estar en cero. Por lo tanto, se optó por enviar los 12 bits de datos precedidos por 4 bits en valor bajo.

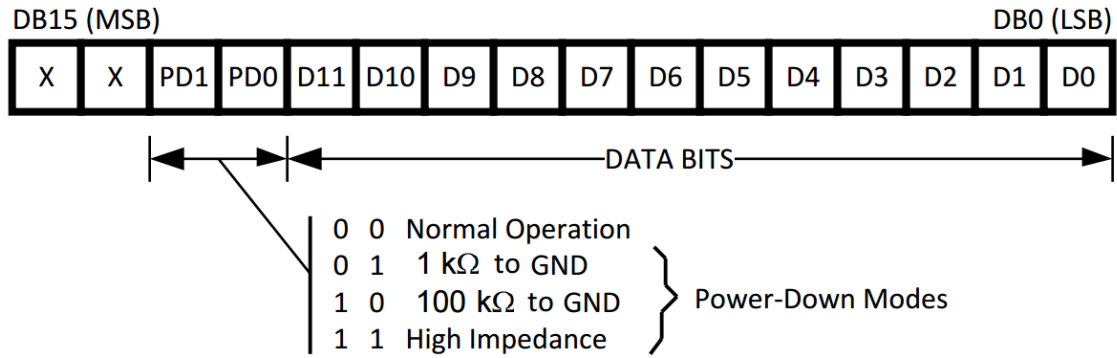


Figura 2.15: Formato de los datos enviados al PmodDA2 para ser convertidos.
Fuente: PmodDA2™ Reference Manual [11].

2.6. Reformulación de los objetivos del proyecto

Una vez analizadas las características del hardware a utilizar para la realización del proyecto, se confeccionó un diagrama en bloques simplificado del emisor. El diagrama presentado en la Figura 2.16 incluye los principales bloques a ser diseñados para garantizar el cumplimiento de los objetivos inicialmente planteados (sección 1.2).

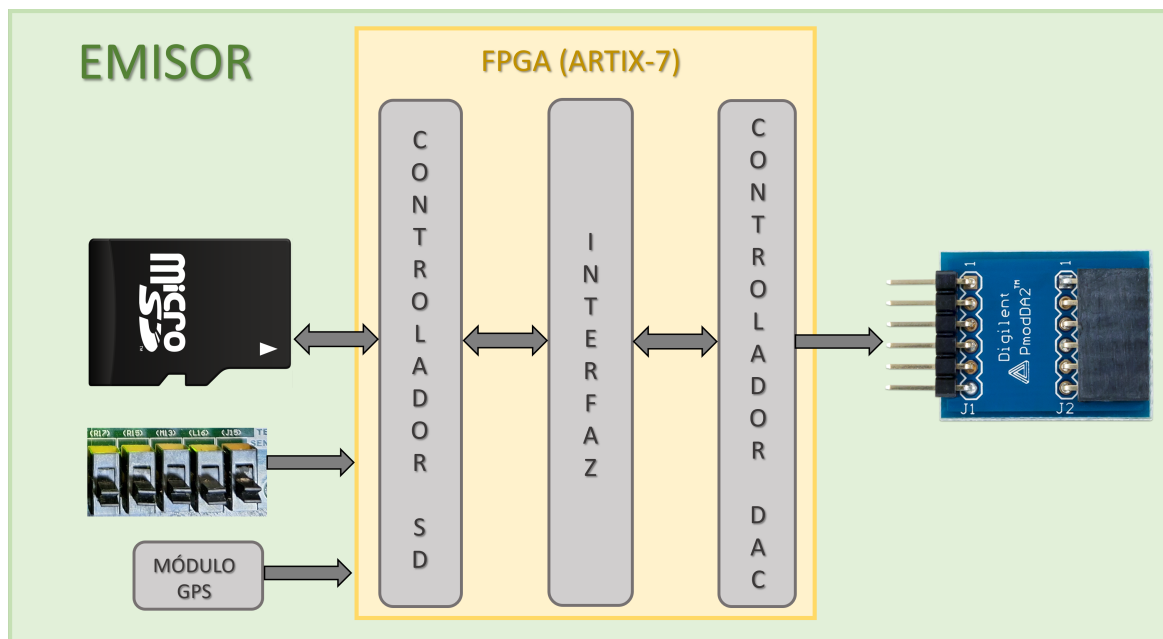


Figura 2.16: Diagrama en bloques simplificado del emisor a ser diseñado.
Fuente: Elaboración propia.

El proyecto queda entonces delimitado al diseño, simulación e implementación de tres grandes bloques funcionales en la placa de desarrollo:

- **Controlador de memoria SD:** Debe permitir la lectura y manipulación de los datos de la memoria microSD, respetando sus características físicas,

presentadas en la sección 2.4.

- **Controlador del DAC:** Debe manipular datos digitales para enviarlos al PmodDA2 en el formato adecuado para su conversión, según los conceptos desarrollados en la sección 2.5.
- **Interfaz entre los controladores:** Debe permitir la transferencia de datos entre los dos bloques controladores.

Los bloques mencionados se implementan por medio del lenguaje VHDL, haciendo uso de los recursos de la placa de desarrollo Nexys 4 presentados en la sección 2.3. En los siguientes capítulos se describe el diseño de los tres bloques.

3. Lectura de la memoria SD

Para facilitar la utilización del dispositivo emisor, las señales codificadas y moduladas se almacenan en una memoria SD que luego es leída por la FPGA. Para ello, es necesario realizar el diseño de un controlador de memoria SD con su respectivo host, tal como se presenta en el diagrama de bloques simplificado en la Figura 3.1. Asimismo, se debe normalizar el formato de las señales a almacenar para el controlador diseñado.

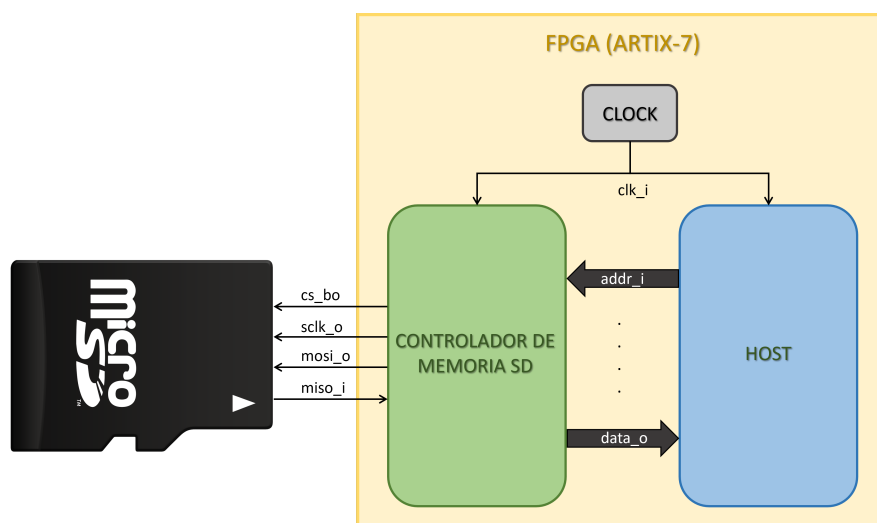


Figura 3.1: Esquema básico del controlador de memoria, la memoria SD y el host.
Fuente: Elaboración propia.

En el siguiente capítulo se describen los pasos a seguir para guardar las señales en la memoria en un formato compatible con el controlador y se realiza una descripción del controlador de memoria diseñado. Para ello se parte de un diagrama de bloques inicial que se va modificando a medida que se avanza con el diseño. Asimismo, se realizan simulaciones para verificar el funcionamiento del controlador, para ello, se implementa un conjunto controlador-host simplificado que permite verificar el funcionamiento del controlador interactuando con una memoria micro SD real.

3.1. Almacenamiento de las señales codificadas y moduladas en la memoria SD

3.1.1. Formato de los datos

Previo al diseño del controlador, se definió de qué manera se almacenarían en la memoria las señales a transmitir. Esta condición determinaría la complejidad del

controlador a implementar. Por lo tanto, para optimizar el diseño, se decidió realizar una lectura de la memoria accediendo a sus bloques elementales. Asimismo, para evitar la adición de un bloque de procesamiento y conversión de los datos leídos, se eligió guardar los datos en un formato *.bin*. De esta manera, cada posición de memoria leída contiene la información digital a transmitir.

Dado que el conversor PmodDA2 admite palabras de 12 bits, los valores instantáneos de las señales a transmitir se codifican en palabras binarias de 12 bits. Por consiguiente, cada valor instantáneo de la señal ocupa un byte y medio de memoria. Teniendo en cuenta que la lectura de datos se realiza de a bytes, se colocan 4 ceros al inicio de cada valor instantáneo de señal, obteniéndose una representación de la amplitud en palabras de 16 bits. De esta manera, al acceder a dos bytes de memoria consecutivos ya se cuenta con la amplitud en valor binario de la señal a ser emitida en un instante del tiempo.

3.1.2. Acceso a las distintas señales codificadas y moduladas

Una vez definido el formato de los datos, se analizó la forma en que se accedería a las señales codificadas y moduladas (de la sección 2.2) para así transmitir la señal deseada.

En primera instancia, se consideró utilizar un sistema de File Allocations Table (FAT) para buscar la señal escogida a partir del nombre del archivo, obtener las direcciones en las cuales se almacena y leer el contenido de dichas direcciones. No obstante, la complejidad del controlador de memoria SD se incrementaba significativamente. Dado que para la aplicación particular las señales almacenadas tendrán características similares de duración y amplitud, no resulta de gran utilidad implementar un bloque de búsqueda de archivos por nombre. Esto se resuelve al definir un límite en el tamaño máximo de la señal a transmitir, y la cantidad de señales almacenadas a la vez.

Para simplificar el diseño del modulo que se encarga de leer los datos de la memoria, se limitó a un máximo de 4 señales almacenadas, correspondiente a cada uno de los esquemas estudiados. Además el almacenamiento máximo de cada señal no puede superar los 74 kBytes (128 bloques de 512 bytes), lo cual equivale a un máximo de 37888 muestras.

Para comprender la forma en que la memoria almacena los datos, se realizaron una serie de pruebas empleando diferentes tipos de archivos. Luego de distintas pruebas de almacenamiento de archivos de texto y binarios en una memoria SD, utilizando el software libre de edición hexadecimal HxD ¹, se llegó a la conclusión de que el contenido de los archivos siempre se almacenaba a partir de la misma dirección de memoria. Por lo tanto, conociendo el tamaño del archivo, la lectura de su contenido puede realizarse apuntando a la dirección inicial de almacenamiento. Como consecuencia se optó por almacenar en un único archivo los valores de las 4 señales en forma consecutiva, para así también asegurar que no existan posiciones de memoria vacías entre señales.

¹<https://mh-nexus.de/en/hxd/>

Por último, cada señal codificada y modulada debe tener la misma cantidad de muestras, para así simplificar aún más la lógica del controlador. En caso de que se requiriera una menor cantidad de muestras, las posiciones restantes de la memoria se completan con ceros, tal que la salida del emisor sea nula.

3.1.3. Instrucciones para guardar las señales en la memoria

Para almacenar los valores instantáneos de las señales a emitir en la memoria SD, se comienza por convertir su valor de amplitud a formato hexadecimal de 4 caracteres. Para ello se implementa el código 3.1 en Matlab (o software equivalente)². Como resultado se obtiene un vector de muestras que aprovecha la máxima excursión de voltaje del PmodDA2 y que se muestreó a la frecuencia adecuada (1 MSa/s). Los datos resultantes se encuentran en formato string y cada dato está encerrado entre comillas.

Código 3.1: Obtención de la representación hexadecimal de las muestras de la señal codificada y modulada.

```
1 %% Representación de las señales codificadas y moduladas en palabras
   ↳ hexadecimales de 4 caracteres.
2
3 % Como el PmodDA2 tiene una resolución de 12 bits la máxima excursión de
   ↳ voltaje se obtiene con la palabra max_rep
4
5 max_rep = 2^12-1;
6
7 % En y se coloca el vector de datos de la señal codificada y modulada.
8
9 y = signal(1,:);          % Modificar según corresponda
10
11 % fs es la frecuencia de muestreo con la que se obtuvieron las muestras de
   ↳ y.
12
13 fs = 500e3;                % Modificar según corresponda
14 fdes = 1e6;                % Frec. de muestreo del PmodDA2
15
16 % Resample de la señal y
17
18 y_resampled = resample(y,fdes/fs,1);
19
```

²Este código fue diseñado para señales con portadoras senoidales de valor máximo 1 y valor mínimo -1.

```

20 % Saturación del vector de datos resampleado
21
22 for i=1:length(y_resampled)
23     if y_resampled(1,i)>1
24         y_resampled(1,i) = 1;
25         i
26     else if y_resampled(1,i)<-1
27         y_resampled(1,i) = -1;
28         i
29     end
30 end
31 end
32
33 % Se calcula la excursión de la señal para aprovechar todo el rango de
    ↪ representación del PmodDA2
34
35 min_val = min(y_resampled);
36 max_val = max(y_resampled);
37
38 exc = max_val - min_val;
39
40 % Se lleva el valor mínimo de la señal a 0
41
42 fun_wo_cc = y_resampled + abs(min_val);
43
44 % Se obtienen los valores decimales de la representación en 12 bits
45
46 fun_integers = fun_wo_cc.*max_rep/exc;
47
48 fun_integers1 = round(fun_integers);
49
50 % Se convierten los valores decimales a palabras hexadecimales de 4
    ↪ caracteres
51
52 hex_data = dec2hex(fun_integers1,4);

```

El siguiente paso consiste en copiar el resultado obtenido en la consola y pegarlo en un editor de textos. Para este caso se utilizó el software Notepad++³. A continuación, se eliminan todas las comillas ('), los retornos de carro (\r) y los

³<https://notepad-plus-plus.org/downloads/>

retornos de línea (\n). En la Figura 3.2 se presenta como ejemplo la eliminación de los retornos de carro.

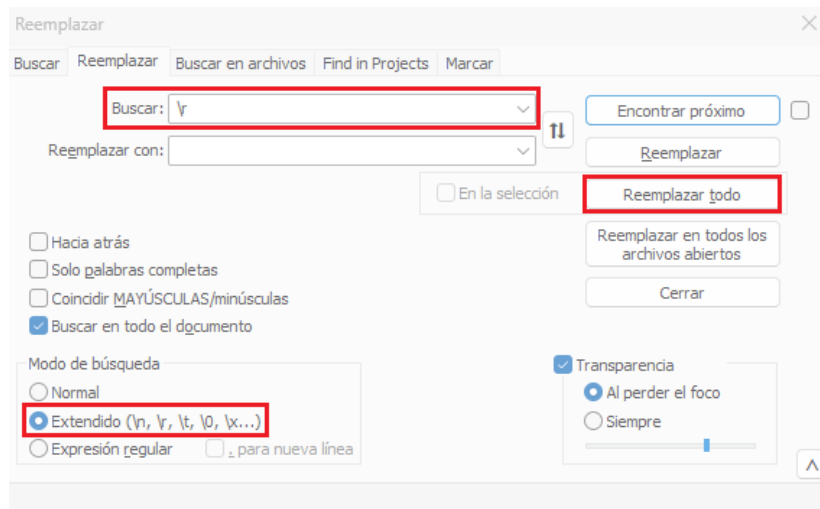
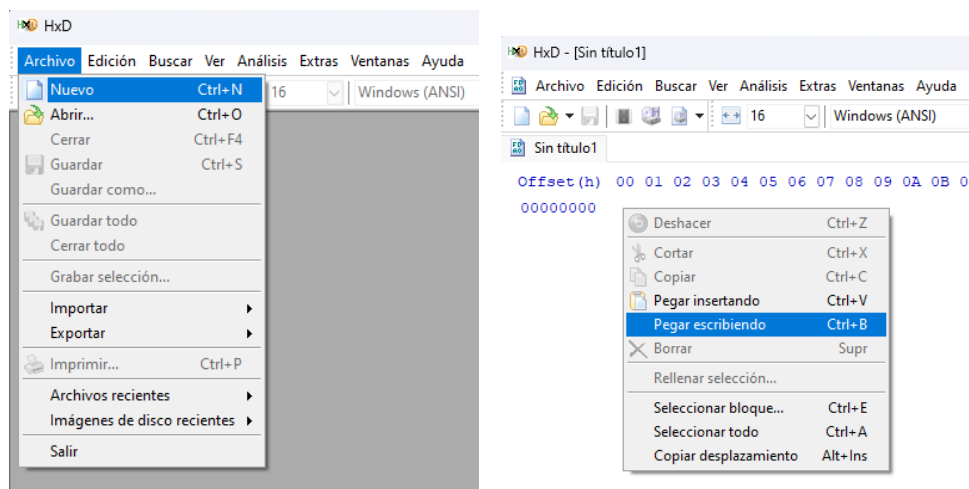


Figura 3.2: Eliminación de los retornos de carro en el software Notepad++.
Fuente: Elaboración propia.

A continuación, se genera en el software HxD un archivo binario (Figura 3.3(a)) y se pega escribiendo los valores hexadecimales copiados (Figura 3.3(b)).



(a) Generación del archivo binario

(b) Inserción de los datos copiados

Figura 3.3: Utilización del editor hexadecimal HxD para la creación del archivo binario de datos.

Fuente: Elaboración propia.

El procedimiento descrito se repite para cada una de las señales codificadas y moduladas, pegando cada señal sobre un único archivo binario. Es fundamental que el usuario recuerde el orden en el cual pega las señales en el archivo binario para así saber el orden de almacenamiento en la memoria.

3.2. Controlador de memoria SD

3.2.1. Esquema básico del sistema a diseñar

Tal como se mencionó en la Sección 2.4 para interactuar con la memoria SD se necesita un host que envíe los comandos de la rutina de inicialización y que solicite los datos que necesita obtener de la memoria. Si bien ambas operaciones quedan contempladas dentro de las tareas del host, resulta conveniente obtener al menos dos bloques separados que operen coordinadamente, siendo la función de uno realizar operaciones elementales de lectura y la del otro solicitar los datos a leer y procesarlos en función de su aplicación. De esta manera, se pueden distinguir dos bloques diferenciados: el controlador de la memoria SD y la lógica de aplicación del host. En la Figura 3.4 se presenta el esquema básico del sistema a diseñar con los bloques mencionados.

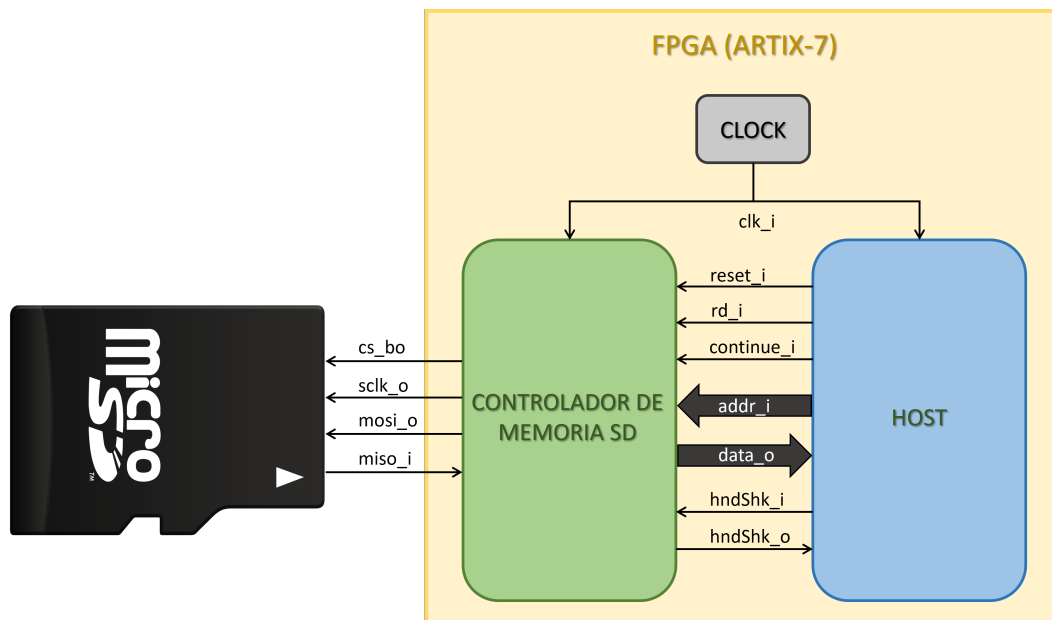


Figura 3.4: Esquema del sistema Controlador - Host.

Fuente: Elaboración propia.

Las funciones del controlador están vinculadas a la inicialización de la memoria para la operación en el modo SPI, es decir, enviar los comandos y verificar las respuestas descritas en la sub-sección 2.4.4. Adicionalmente, el controlador se encarga de la lectura de los bloques elementales de 512 bytes de la memoria. Por otro lado, la función del host es indicarle al controlador la dirección de memoria a la cual debe acceder, y de la cual extraerá los datos a procesar.

Las principales conexiones entre los bloques mencionados se listan a continuación:

- **reset_i:** Entrada activa alta que controla a los circuitos POR (Power On Reset) e inicializa las máquinas de estados del controlador.

- **rd_i**: Entrada activa alta que el host utiliza para indicarle al controlador que se debe comenzar un proceso de lectura.
- **data_o**: Es un Bus de 8 bits a través del cual el controlador le envía a la lógica de aplicación del host los bytes leídos de la memoria.
- **hndShk_i** y **hndShk_o**: Son puertos que se utilizan para garantizar el correcto intercambio de la información leída, entre el controlador y el host.

3.2.2. Diseño del controlador de memoria SD

Máquina de estados global del controlador

Para el diseño del bloque de control de memoria en la FPGA, se generó un proceso con un reloj de 100 MHz como única señal en su lista de sensibilidad. Este proceso, que es activado por el flanco ascendente de la señal de reloj, cuenta con un condicional que se subdivide en seis estados con diferentes acciones a realizar. La ejecución de estas acciones está determinada por el valor de las señales de handshake (**hndShk_i** y **hndShk_o**), el valor de la entrada de reset (**reset_i**) y el estado anterior. En otras palabras, los seis estados componen una máquina de estados donde el estado actual depende tanto del estado anterior como de las entradas. Estas pueden ser las salidas del host o las salidas de la máquina de estados principal o FSM (Finite State Machine) principal. En la Figura 3.5 se presenta la máquina de estados con sus seis estados.

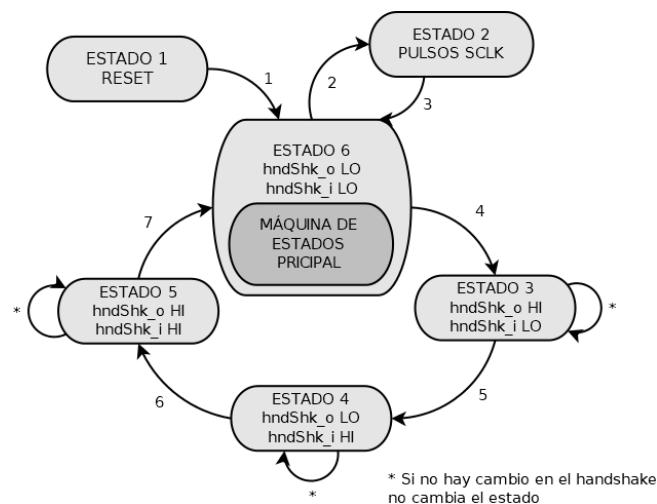


Figura 3.5: Máquina de estados general del controlador de memoria SD.

Fuente: Elaboración propia.

Los seis estados del proceso se describen a continuación:

1. **RESET**: Si la entrada de reset se encuentra activa, se setea como estado inicial de la máquina de estados del estado 6, el estado **START_INIT**, que se detallará más adelante.

2. **Contador del reloj SCLK distinto de cero:** En este estado, se disminuye un contador con cada flanco activo de clock hasta llegar a cero. Esto tiene como objetivo enviar pulsos de reloj a la memoria SD para permitir que esta realice distintas tareas al pasar de un estado al otro (en caso de que fuera necesario).
3. **Handshake del lado del controlador en ALTO (hndShk_o), y en BAJO del lado del host (hndShk_i):** El controlador ha manifestado la disponibilidad de un dato leído en la salida **data_o**. En este caso no se realizan operaciones hasta recibir un cambio en el handshake del host.
4. **Handshake en ALTO en ambos lados:** El host indica que ha recibido el dato enviado por el controlador, por lo tanto el controlador baja su handshake y espera a que el host haga lo mismo.
5. **Handshake en BAJO en el lado del controlador (hndShk_o), y ALTO en el lado del host (hndShk_i):** En este estado no se realiza ninguna acción, solo se espera a que el host baje su handshake.
6. **Handshake en BAJO en ambos lados:** El controlador está listo para operar. Dentro de este punto se incluyó una máquina de estados de tipo Mealy tal que la salida dependiera tanto del estado actual como de las entradas. Esta máquina de estados además de permitir la lectura de la memoria, se encarga de realizar la inicialización de la misma, según lo descrito en la subsección [2.4.4](#). Los cambios en las señales y variables de la máquina de estados presente en este punto tiene como consecuencia el paso del estado 6 a los estados 2, 3, 4 o 5 (el estado 1 se accede por medio del reset).

Máquina de estados principal del controlador

A continuación se describen los estados de la máquina de estados principal (contenida dentro del estado 6 de la máquina de estados global) del controlador.

- **START_INIT:** Se prepara a la máquina de estados para la inicialización a partir de la configuración inicial de las salidas, señales y variables internas del controlador. Se fija la frecuencia de **SCLK** en el valor de inicialización (400 kHz) y se configura al controlador para que envíe 160 pulsos de clock a la memoria (ver subsección [2.4.4](#)). Por último se actualiza el valor del estado a **SEND_CMD0**.
- **DESELECT:** en este estado se deselecciona la memoria SD colocando la entrada CS en ALTO y se envían pulsos de reloj.
- **PULSE_SCLK:** incluye un contador para enviar una determinada cantidad de pulsos de reloj y al finalizar se actualiza el estado siguiente al estado reservado.
- **SEND_CMD0:** Se habilita la memoria SD (se coloca CS en BAJO), se la deselecciona, se actualiza la variable a transmitir (comando CMD0) y el estado a TX.BITS, y se reserva el estado **SEND_CMD8**.
- **START_TX:** Se comienza la transmisión del comando a enviar colocando el reloj SCLK en estado bajo y el MSB del comando en la salida **mosi_o**.

- **TX.BITS:** en este estado se envían a la memoria SD los comandos a ser ejecutados y las direcciones de los bloques de memoria a leer. Por lo tanto, la máquina de estados ejecuta este estado tanto al enviar los comandos CMD0, CMD8 y ACMD41 durante la inicialización, así como también al enviar el comando de lectura CMD17. Incluye un contador para llevar registro de los bits que quedan por transmitir y cuando finaliza la transmisión actualiza el estado siguiente. Si el comando enviado no requiere respuesta el estado siguiente se actualiza al estado almacenado. Por otro lado, si el comando requiriera respuesta, se actualiza el estado siguiente a **GET_CMD_RESP**.
- **GET_CMD_RESP:** se encarga de recibir la respuesta de la tarjeta SD al comando CMD0.
- **RX.BITS:** se reciben los bits enviados por la memoria al controlador y cuenta con un contador que llega a cero al recibirse un byte completo. Dentro de este estado se pueden recibir respuestas de la memoria a algún comando enviado por el controlador, como por ejemplo el CMD8. En este caso, como el resultado de la respuesta no es relevante (lo importante es recibir algún tipo de respuesta por parte de la memoria SD, ver 2.4.4) se actualiza el estado a partir del estado de reserva (almacenado en el estado anterior). Asimismo, dentro de este estado se reciben los datos que el controlador ordena leer de la memoria. En este caso, como la respuesta recibida debe ser enviada al host, cuando el contador llega a cero, se coloca el dato recibido en la salida paralela del controlador y se pone en alto la salida del handshake **hndShk_o**.
- **CHK_CMD0_RESP:** Se verifica que la respuesta de la memoria al comando CMD0 sea la adecuada. En caso de serlo, se actualiza el estado actual al estado reservado (CMD8).
- **SEND_CMD8 :** La secuencia es similar a la del estado **SEND_CMD0** con la diferencia de que el comando a enviar es el 8. Se reserva el estado **GET_CMD8_RESP**.
- **GET_CMD8_RESP :** Se verifica que la memoria haya generado una respuesta al comando CMD8. En ese caso se actualiza el estado a **RX.BITS** y se reserva el estado CMD55.
- **SEND_CMD55:** Se actualiza la variable a transmitir al comando 55 y se reserva el estado **SEND_CMD41**, dado que este comando es un preámbulo.
- **SEND_CMD41:** Se actualiza la variable a transmitir al comando 41 y se actualiza el estado de reserva a **CHK_ACMD41_RESP**.
- **CHK_ACMD41_RESP:** Se verifica que la memoria haya salido del estado inactivo luego del comando ACMD41, comparando el byte recibido con h00. En ese caso se actualiza el estado a **WAIT_FOR_HOST_RW**. Por otro lado, si la memoria continúa en el estado inactivo se actualiza el estado a **SEND_CMD55** para volver a enviar el comando ACMD41.
- **WAIT_FOR_HOST_RW:** La función general de este estado es esperar a que el host envíe una orden de lectura al controlador de memoria SD, activando la salida **rd_i**. En este estado se carga en los bits a transmitir el comando de

lectura junto con la dirección del bloque a leer. Si la lectura a realizar involucra múltiples bloques, se recurre a este estado cada vez que finaliza la lectura de un bloque y se actualiza la dirección del bloque a leer.

- **RD_BLK**: Se espera el Token de inicio de datos que precede a un bloque de 512 bytes enviado por la memoria SD al comienzo de una operación de lectura, y se actualiza el estado siguiente a **TX BITS**. El estado **RD_BLK** cuenta además con un contador que verifica que se haya leído el bloque completo para posteriormente recibir 2 bytes de CRC que son enviados por la memoria (pero que no deben ser enviados del controlador al host).
- **REPORT_ERROR**: a esta rutina se acude en caso de que ocurra algún error y se colocan los bits recibidos en la salida de error (**error_o**).

De los estados mencionados, existen algunos que en caso de cumplirse las condiciones ideales (sin errores) de inicialización y transmisión de datos, solo son ejecutados una vez (por ejemplo **SEND_CMD0**) mientras que hay otros a los que se regresa continuamente (por ejemplo **TX BITS**). Como consecuencia, el diagrama de estados completo de la máquina de estados resulta ser de gran complejidad. Sin embargo esto permite una optimización del código resultante.

En la Figura 3.6 se presenta un diagrama de flujo simplificado de la rutina de inicialización de la máquina de estados principal del controlador de memoria SD, con los estados explicados anteriormente, y su respectiva leyenda. Para simplificar el diagrama se agruparon en dos funciones (**TX_AND_RX** y **SEND_PULSES**), un conjunto de estados que se repiten en varias oportunidades dentro del diagrama de flujo. La descripción de dichas funciones se encuentra en la Figura 3.7.

La rutina de inicialización se ejecuta únicamente luego de insertar la memoria SD en la ranura que contiene la placa de desarrollo Nexys 4 DDR, y después de resetear el controlador de memoria. Por lo tanto, si la inicialización es exitosa, el controlador y la memoria quedan en condiciones de intercambiar datos almacenados en la misma y no se vuelve a los estados indicados bajo el nombre de “estados de inicialización”. Por otro lado, los estados indicados como “estados genéricos” pueden encontrarse tanto en la inicialización como en el proceso de lectura.

En la Figura 3.8 se presenta un diagrama de estados de una rutina de lectura de un bloque de 512 bytes. Asimismo, en la Figura 3.9 se presenta la secuencia de estados representada bajo el nombre **RX_BYTES**, que consiste en la reiteración del estado **RX BITS** hasta que se haya recibido un byte completo y la actualización del estado a **RD_BLK**. Se puede observar que tal como se indica en la Figura 3.8, la rutina de lectura contiene 3 fases diferenciadas:

- **Recepción del Token de inicio** de bloque, el cual es enviado por la memoria SD previo al envío del primer byte del bloque.
- **Lectura del bloque de memoria** de 512 bytes, donde se obtiene la información que realmente se desea obtener al leer la memoria.
- **Recepción de 2 bytes de CRC** a partir de los cuales se confirma la correcta lectura del bloque.

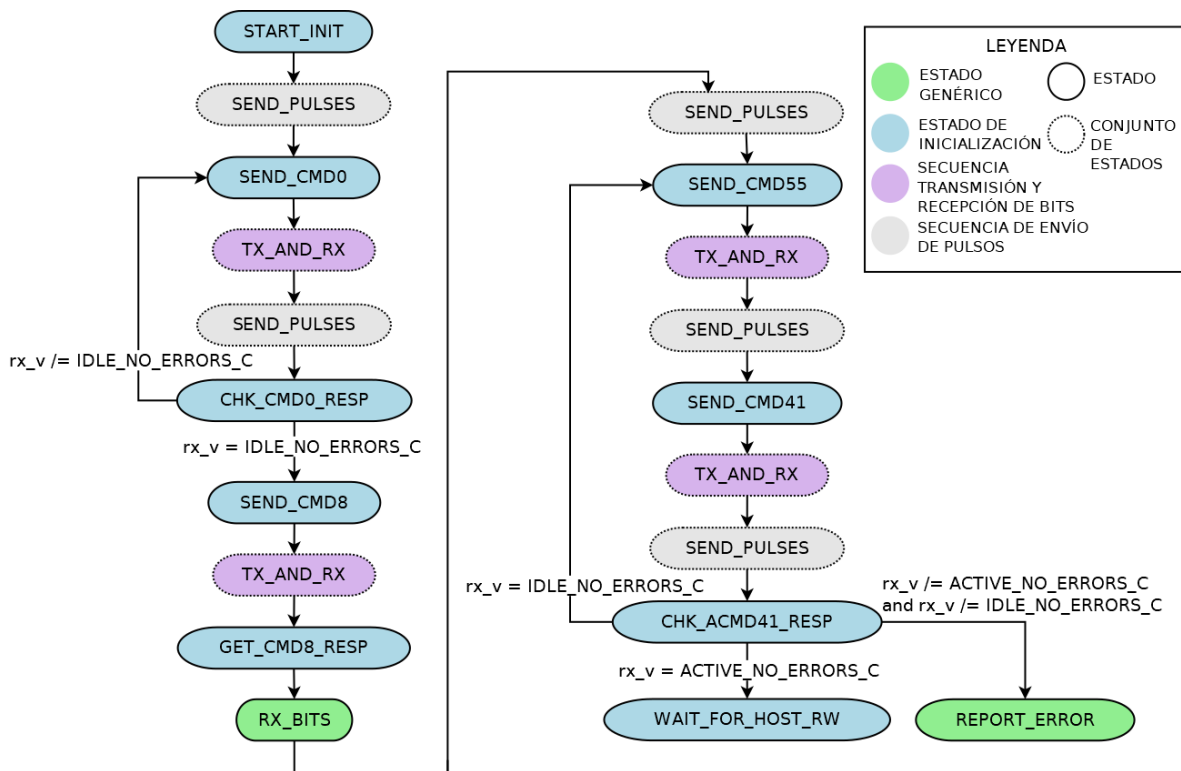


Figura 3.6: Diagrama de flujo simplificado de la rutina de inicialización del controlador de memoria SD.

Fuente: Elaboración propia.

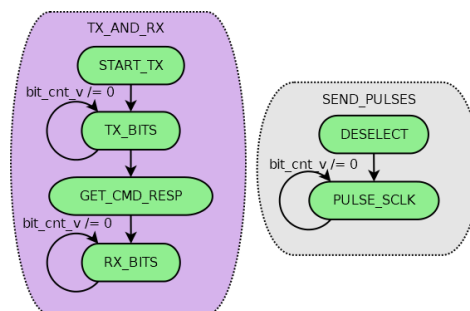


Figura 3.7: Funciones TX_AND_RX y SEND_PULSES.

Fuente: Elaboración propia.

Cabe destacar que dentro de la lectura de los 512 bytes, se produce el proceso de handshake entre el controlador y el host, que es parte de la máquina de estados global del controlador (3.2.2). Esta operación controla que la memoria no envíe otro byte al controlador antes de que el host haya recibido el último byte enviado por el controlador. De esta manera, todos los datos leídos de la memoria son recibidos correctamente para ser procesados o utilizados por el host.

Cuando se han ejecutado todos los estados comprendidos entre TX_AND_RX y SEND_PULSES se completa la lectura de un bloque de memoria. Se regresa entonces al estado WAIT_FOR_HOST_RW y en el caso de que la entrada **rd.i** continúe en alto, se comienza la lectura del bloque siguiente. Para ello se aumenta previamente la variable **addr.v**.

Cuando el proceso de lectura finaliza, la entrada **rd.i** pasa a un valor bajo por lo cual la máquina de estados regresa continuamente al estado **WAIT_FOR_HOST_RW**. Allí espera un cambio en el valor de **rd.i** que representa una nueva solicitud de lectura.

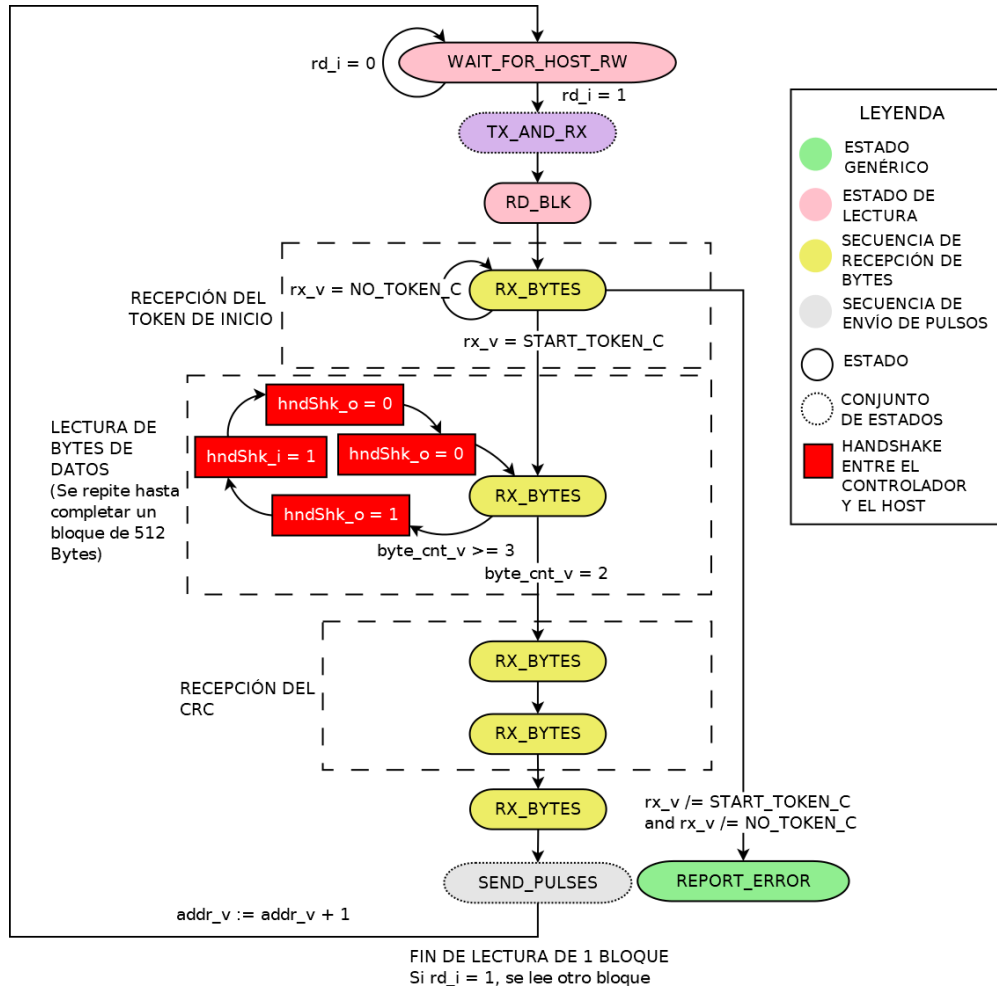


Figura 3.8: Diagrama de flujo simplificado de la rutina de lectura de un bloque de memoria.

Fuente: Elaboración propia.

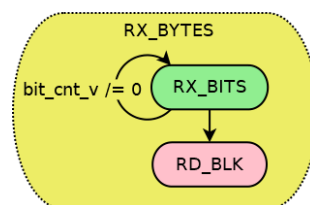


Figura 3.9: Funciones RX_BITS y SEND_PULSES.

Fuente: Elaboración propia.

3.2.3. Simulación del bloque controlador de memoria SD

Para verificar el correcto funcionamiento del bloque diseñado se creó un proyecto con el controlador y un banco de pruebas. El objetivo era verificar las rutinas de inicialización y lectura descritas, lo cual requeriría el diseño de un emulador de memoria SD en VHDL. Dada la complejidad implicada en el diseño de un módulo que emule las respuestas de una memoria SD, se optó por introducir, de forma manual, las respuestas esperadas por la memoria, siguiendo la hoja de datos del fabricante. Dentro del banco de pruebas se introdujo el reloj de 100 MHz, se diseñó un proceso que emulara el handshake con el controlador y se generó una secuencia de bits aleatorios para colocar en la entrada de datos del controlador (emulando la salida de datos de la memoria y por ende la entrada de datos del controlador **miso_i**). La entrada al controlador **rd_i** se seteó en alto, se colocó una dirección ficticia en la entrada de dirección del controlador **addr_i** y se configuró un pulso de reseteo para asegurar el inicio de la secuencia de inicialización.

En la Figura 3.10 se presenta el resultado de la simulación temporal del controlador de memoria. Se observa que los bytes que ingresan serialmente al controlador por la entrada **miso_i** son cargados en la salida paralela **data_o** luego del handshake (flecha en la figura). Es importante destacar que para llegar a la rutina de lectura se debió superar exitosamente la rutina de inicialización, para lo cual se introdujeron manualmente, en la consola del simulador, las respuestas esperadas a los comandos de inicialización CMD0, CMD8, ACMD41 y al Token de inicio de bloque.

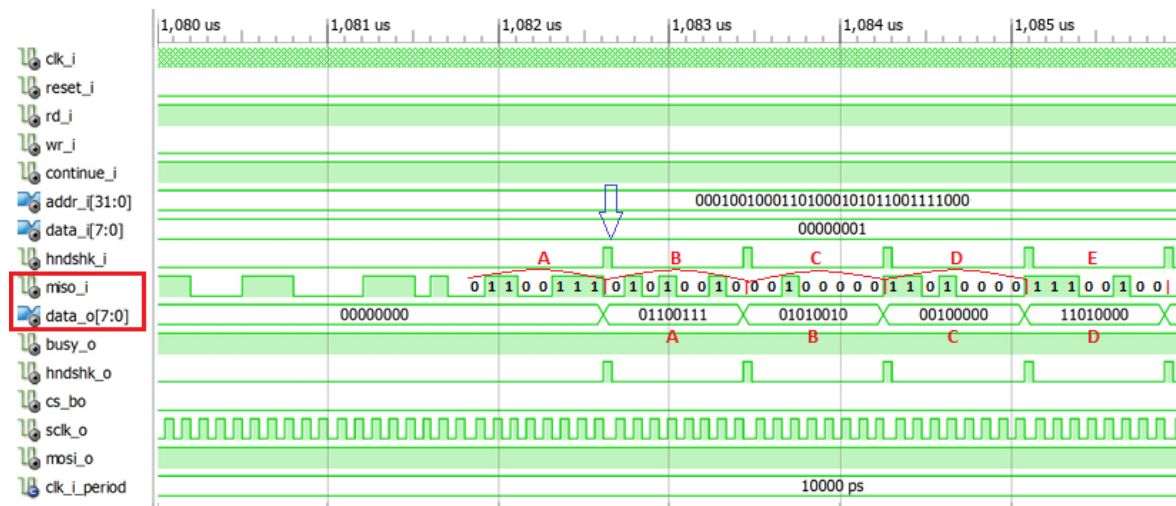


Figura 3.10: Simulación del controlador de memoria.

Fuente: Elaboración propia.

3.2.4. Implementación del bloque controlador de memoria SD

Una vez verificada la funcionalidad del controlador por medio de simulaciones se creó un proyecto nuevo en Vivado para realizar una implementación en placa y evaluar la interacción entre el controlador y una memoria SD real. Para ello, se

debió añadir un host que realizara el proceso de handshake con el controlador y permitiera la lectura de datos (en la simulación el handshake se realizaba en el banco de pruebas). El objetivo de esta primera implementación se orientó a la verificación de la correcta inicialización de la memoria y la lectura de bytes. Cabe aclarar, que la verificación de la lectura se orientó a la acción de leer bytes sin importar su contenido.

Para validar el diseño, era necesario conocer en qué estado se encontraba la máquina de estados. Para ello, se necesitaba algún tipo de indicador de lo que estaba sucediendo a nivel del controlador y el host. Teniendo en cuenta la gran cantidad de leds con los que cuenta la Nexys 4 DDR se optó por encender leds al pasar por distintos estados. Sin embargo, al contar con más estados que leds, se debió repetir el proceso una gran cantidad veces, cambiando los estados a representar con los leds. Finalmente se logró asegurar la secuencia correcta de estados, tal como lo indican las máquinas de estados de la subsección [3.2.2](#).

En la Figura [3.11](#) se presenta una de las implementaciones realizadas con los leds 4, 8, 9, 10, 12, 15 (verdes), 16 y 17 (azul y rojo respectivamente) encendidos. A continuación se presenta el significado de cada led:

- LED0 : START_INIT
- LED1 : SEND_CMD0
- LED2 : SEND_CMD8
- LED3 : CHK_ACMD41
- LED4 : WAIT_FOR_HOST_RW
- LED5 : RD_BLK
- LED6 : byte_cnt.3
- LED7 : byte_cnt.2
- LED8 : byte_cnt.0
- LED9 : TX_BITS
- LED10 : RX_BITS
- LED11 : DESELECT
- LED12 : PULSE_SCLK
- LED13 : REPORT_ERROR
- LED14 : No utilizado.
- LED15 : rd_i
- LED16 : hndShk_o
- LED17 : hndShk_i

A la frecuencia de reloj empleada (100 MHz), hay estados que permanecen activos durante un tiempo muy breve, por lo cual los leds que los representan se

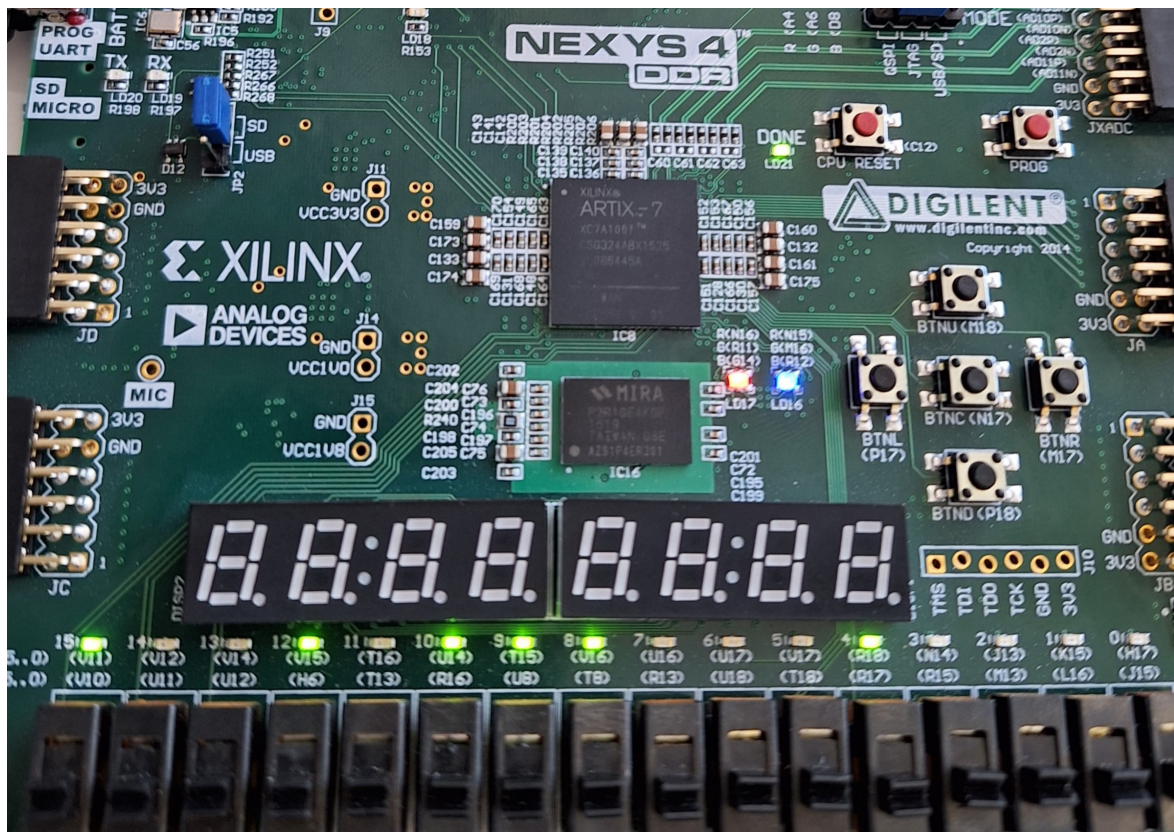


Figura 3.11: Implementación del controlador de memoria en la Nexys 4 DDR para la verificación del controlador.

Fuente: Elaboración propia.

encienden y apagan al instante. Los primeros 4 leds (del LED0 al LED3) que se encienden cuando la máquina de estados está ejecutando la rutina de inicialización, solo permanecen encendidos unos pocos pulsos de reloj luego de accionar el reset (por medio de un interruptor). Por ello, si bien la inicialización se realiza, no se observan los leds mencionados en estado encendido en la Figura 3.11. De hecho, los leds que si están encendidos en la figura, en realidad están constantemente encendiéndose y apagándose pero a la frecuencia del reloj se ven continuamente encendidos. De esta manera, se observa que el controlador está leyendo posiciones de memoria, dado que se está realizando de forma continua el handshake entre el controlador y el host (LED16 y LED17), y se ejecuta también la transmisión y recepción de bits. Asimismo, se observa que el controlador envía pulsos de reloj (de 25 MHZ) a la memoria (LED12) y ejecuta sostenidamente el estado **WAIT_FOR_HOST_RW** para actualizar la dirección del bloque a leer. De los leds utilizados para representar los estados el único que nunca se encendió durante la prueba de implementación fue el 13, que representa al estado **REPORT_ERROR**.

En conclusión, esta implementación permitió asegurar que el controlador diseñado supera correctamente la rutina de inicialización y realiza operaciones de lectura de bytes. Sin embargo, no permite garantizar que los bytes leídos por el controlador sean los que están almacenados en la memoria SD.

3.3. Bloque host para la puesta a prueba del controlador

Teniendo en cuenta las conclusiones de la sección anterior (3.2.4), se realizó un diseño que permitiera verificar la correcta lectura de la memoria SD a través del controlador. Para ello se implementó un bloque host que cumpliera con las siguientes funcionalidades:

- Apuntar a la dirección de memoria que se desea leer.
- Realizar operaciones de lectura de uno o más bloques.
- Realizar la operación del handshake una vez que se ha recibido el byte enviado por el controlador.
- Procesar los datos recibidos para así verificar la correcta lectura de los datos almacenados en la memoria, comparándolos con una copia local.

En primera instancia se diseñó un bloque host de prueba que leyera los datos de un único bloque de la memoria y se compararan los bytes leídos con un arreglo de datos almacenados en el host. Al verificarse la correcta lectura de un bloque se reconfiguró el host diseñado para leer 10 bloques seguidos y verificar bytes aleatorios comparándolos con los almacenados en el host. En los siguientes apartados se presenta una breve explicación del host de prueba diseñado con sus respectivas simulaciones y la implementación final.

3.3.1. Host para la lectura de 10 bloques consecutivos

En la Figura 3.12 se presenta un diagrama básico del bloque host diseñado para realizar la verificación del controlador a través de una implementación en placa. Dentro del bloque host se pueden distinguir los bloques de detección de flanco y de lectura multibloque (contorno continuo). A su vez, se distinguen 3 procesos (contorno discontinuo): de handshake, de comparación y de estados. Todos los bloques y procesos comparten la misma señal de reloj para asegurar el sincronismo al igual que ocurría con la máquina de estados del controlador de memoria.

El bloque de *detección de flanco de reloj* es un bloque fundamental para el desarrollo del actual proyecto VHDL, y se utiliza para detectar un flanco de cualquier señal. La importancia de este bloque está vinculada a la necesidad de identificar flancos de señales que no sean la señal de reloj. Todos los bloques y procesos presentes se disparan a través del flanco ascendente de la señal de reloj que puede detectarse con la función **rising_edge(clk_i)**. Esta función pertenece al conjunto de funciones definidas en VHDL. No obstante, existen algunos bloques o procesos que requieren la activación con el flanco ascendente de alguna otra señal, como en este caso **hndShk_o**, pero no se permite anidar dos detecciones de **rising_edge()** de señales diferentes. Por lo tanto, se recurre al diseño de un bloque que cumpla con dicha función.

El *detección de flanco de reloj* tiene como salidas el valor de la señal de entrada en el instante actual y en el instante anterior. En el caso de que existiese un flanco

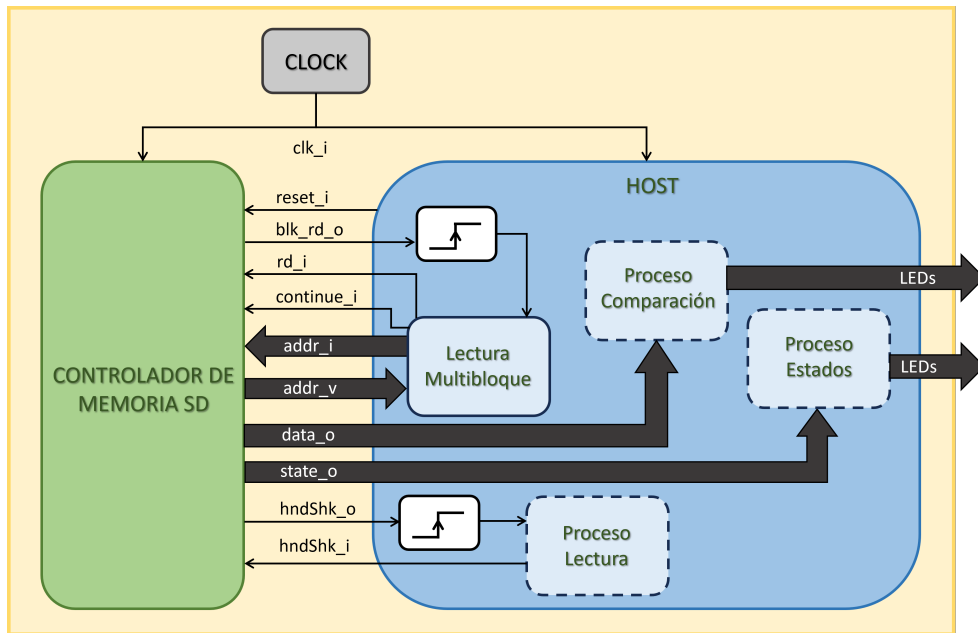


Figura 3.12: Diagrama de bloques del controlador y el host diseñados para la prueba del controlador.

Fuente: Elaboración propia.

creciente de una señal, la salida representativa del instante actual es diferente de la salida representativa del instante anterior. Por lo tanto, detectando una diferencia en el valor binario de las dos salidas, se puede detectar un flanco (ascendente o descendente) en cualquier señal. En el Código 3.2 se puede observar la descripción en HDL del componente detector de flanco.

Código 3.2: Descripción HDL del componente detección de flanco.

```

1  library IEEE, GEN;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use GEN.signalEdgePckg.all;
4
5  entity signalEdge is
6      port(
7          clk_i          : in std_logic;
8          reset_i        : in std_logic;
9          signal_i        : in std_logic;
10         signal_save_o   : out std_logic;
11         signal_save_1_o : out std_logic
12     );
13 end signalEdge;
14
15 architecture arch of signalEdge is

```

```

16
17 signal signal_save    : std_logic;
18 signal signal_save_1 : std_logic;
19
20 begin
21
22 signalEdge_process : process(clk_i,reset_i)
23     begin
24         if reset_i = '1' then
25             signal_save    <= '0';
26             signal_save_1 <= '0';
27         elsif rising_edge(clk_i) then
28             signal_save_1 <= signal_save;
29             signal_save    <= signal_i;
30         end if;
31     end process;
32
33 signal_save_o    <= signal_save;
34 signal_save_1_o <= signal_save_1;
35
36 end architecture;

```

Si bien se denominó a este componente bajo el nombre *detección de flanco de reloj*, en realidad su función es almacenar el valor de la señal de entrada en el instante actual y un tiempo de reloj anterior⁴. Luego, cada componente se encarga de analizar si existe un flanco ascendente o descendente, comparando los valores de las salidas del bloque *detección de flanco de reloj* y testeando el valor actual de la señal. En las descripciones HDL 3.3 y 3.4 se presenta el análisis de las salidas del bloque *detección de flanco de reloj* para la detección de un flanco ascendente y descendente respectivamente.

Código 3.3: Descripción HDL de la detección de un flanco ascendente.

```

1  -- Detección de un flanco ascendente
2  if signal_save_i /= signal_save_1_i and signal_save_i = '1' then
3  -- Comportamiento del circuito al ocurrir un flanco ascendente
4  end if;
5

```

⁴De la misma manera, se presenta en los esquemas el símbolo con la detección del flanco ascendente y descendente, y una única salida. En realidad el componente cuenta con dos salidas, y el componente al que se conectan estas salidas es el que determina la existencia o no del flanco activo.

Código 3.4: Descripción HDL de la detección de un flanco descendente.

```
1  -- Detección de un flanco descendente
2  if signal_save_i /= signal_save_1_i and signal_save_i = '0' then
3  -- Comportamiento del circuito al ocurrir un flanco descendente
4  end if;
5
```

Para la aplicación particular de la implementación del host, el bloque de detección de flanco se utiliza para la detección de los flancos ascendentes de **blk_rd_o**, que indica el fin de lectura de un bloque, y para **hndShk_o**, que es necesario para la ejecución del proceso de lectura. El *proceso de lectura* tiene como funciones principales realizar el handshake con el controlador de memoria y almacenar temporalmente el bloque de 512 bytes que se está leyendo en una memoria de 512 bytes. Básicamente este bloque pasa el valor de la entrada **hndShk_o** a la salida **hndShk_i** al detectar un flanco en la salida de handshake del controlador (**hndShk_o**). En el caso de que este flanco sea ascendente, se incrementa el contador de bytes del proceso y se almacena el dato recibido a través de la entrada **data_o** (proveniente del controlador), en la memoria temporal de 512 bytes. Una vez que el contador llega a 512 se resetea y el nuevo bloque leído se sobrescribe en la memoria temporal de 512 bytes. El diagrama de flujo del proceso descrito se presenta en la Figura 3.13.

En el extracto de código 3.5 se presenta la descripción de la memoria temporal de 512 bytes (**memory_array**) y la inicialización de su contenido en h00. También se presentan los primeros 39 bytes cargados en la memoria **constant_array**, que es un bloque de 512 bytes igual al que se almacena 10 veces en la memoria SD. Para almacenar los datos leídos en la memoria de 512 bytes se actualiza el valor de la señal **mem** en la posición dada por **byte_number**.

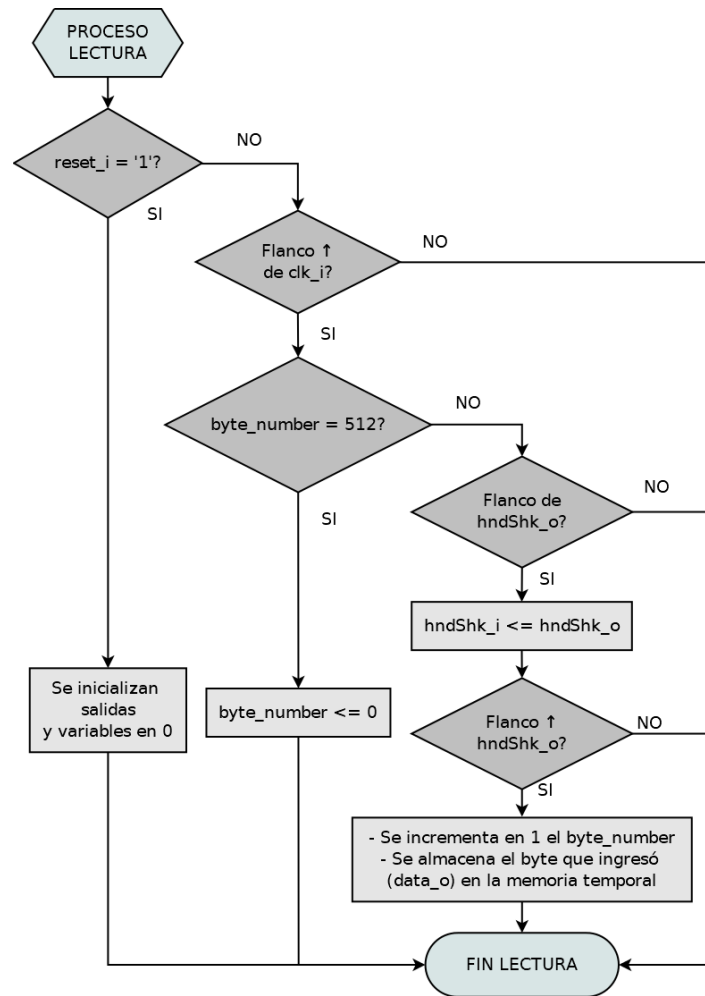


Figura 3.13: Diagrama de flujo del proceso de lectura.
Fuente: Elaboración propia.

Código 3.5: Descripción HDL de la memoria de 512 bytes para la puesta a prueba del controlador.

```

1  --- Memoria de 512 bytes
2  type memory_array is array (0 to 511) of std_logic_vector(7 downto 0);
3  -- Inicialización del contenido de la memoria en h00
4  signal mem : memory_array := (others => (others => '0'));
5  -- Número de byte a leer
6  signal byte_number : integer := 0;
7  -- Memoria de 512 bytes con los bytes a comparar
8  type constant_array is array (0 to 511) of std_logic_vector(7 downto 0);
9  signal const : constant_array := (
10  0 => x"EB", 1 => x"58", 2 => x"90", 3 => x"4D", 4 => x"53", 5 => x"44", 6
    => x"4F", 7 => x"53",

```

```

11 8 => x"35", 9 => x"2E", 10 => x"30", 11 => x"00", 12 => x"02", 13 =>
    ↪ x"40", 14 => x"C8", 15 => x"05",
12 16 => x"02", 17 => x"00", 18 => x"00", 19 => x"00", 20 => x"00", 21 =>
    ↪ x"F8", 22 => x"00", 23 => x"00",
13 24 => x"3F", 25 => x"00", 26 => x"FF", 27 => x"00", 28 => x"00", 29 =>
    ↪ x"20", 30 => x"00", 31 => x"00",
14 32 => x"00", 33 => x"B8", 34 => x"A3", 35 => x"03", 36 => x"1C", 37 =>
    ↪ x"1D", 38 => x"00", 39 => x"00",
15 -- Y así sucesivamente
16 );

```

El bloque de lectura multibloque tiene como función principal indicarle al controlador las direcciones inicial y final de lectura. En la Figura 3.14 se presenta el diagrama de flujo del bloque. Luego compara la dirección de memoria que se actualiza con cada lectura en el controlador, con una dirección almacenada y que se corresponde con la última dirección de bloque a leer. Al alcanzarse la dirección indicada el bloque coloca en bajo las salida **rd_i** indicando que ha culminado el proceso de lectura.

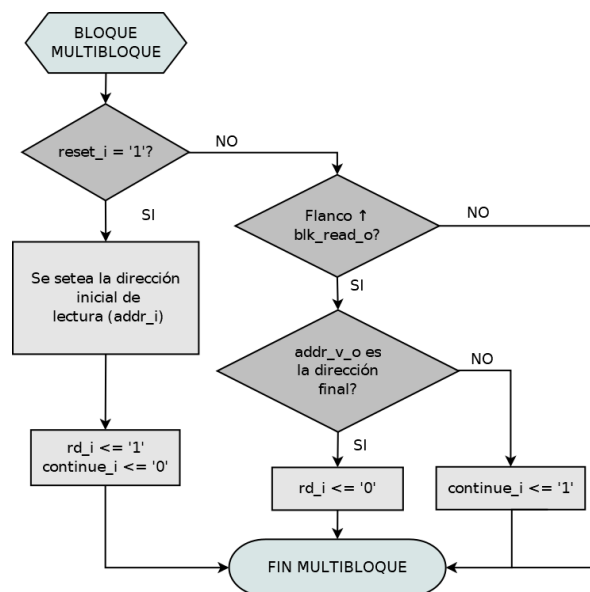


Figura 3.14: Diagrama de flujo del bloque Multibloque.
Fuente: Elaboración propia.

Para verificar que los datos leídos de la memoria sean los almacenados se utiliza el *proceso Comparación*. Este proceso espera a que se termine de leer un bloque dado (testando el valor del contador de bytes del *proceso de lectura*) y compara el valor de bytes aleatorios almacenados en la **memory_array** con los bytes de **constant_array** del código 3.5. Por ejemplo, al terminar de leerse los 512 bytes del bloque h00 de la memoria, se verifica que el byte número 17 de la memoria **memory_array** sea igual al byte número 17 del **constant_array**. Esto se repite para distintos bytes de distintos bloques de memoria. En el caso de que se cumpla la igualdad el *proceso de lectura*

enciende un Led de la Nexys 4 DDR para cada bloque. En la Figura 3.15 se observa el diagrama de flujo para la verificación del byte Y del bloque de memoria X.

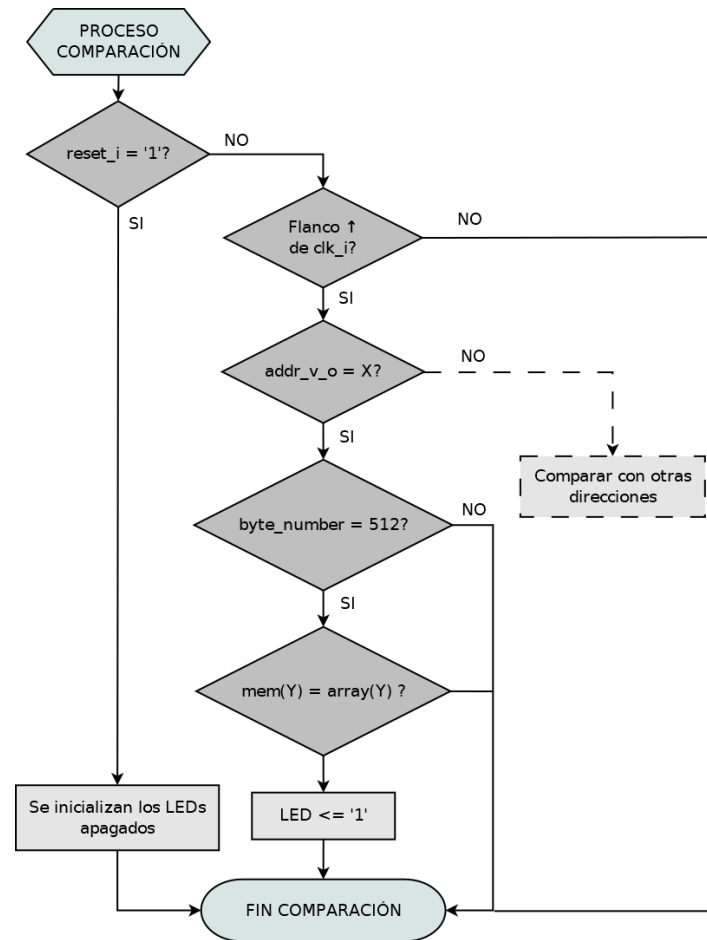


Figura 3.15: Diagrama de flujo del proceso de comparación.

Fuente: Elaboración propia.

Por último, el *proceso Estados* se utiliza para indicar, tal como lo indica su nombre, estados representativos de la máquina de estados del controlador de memoria. Se alimenta de la salida **state_o** proveniente del controlador de estados que se actualiza al cambiar de estados. Inicialmente se realizaron diversas pruebas para depurar el código y verificar que se ejecutaran en forma exitosa la inicialización y las lecturas. Una vez verificadas dichas condiciones, únicamente se conservó la representación por medio de leds del estado **WAIT_FOR_HOST_RW** que indica que el proceso de inicialización culminó exitosamente, y el estado **REPORT_ERROR** que advierte la aparición de algún error durante la lectura o la inicialización.

3.3.2. Simulación para el host para la lectura de 10 bloques consecutivos

En este caso no se realizó una simulación para la verificación del host funcionando en forma conjunta con el controlador puesto que los bloques añadidos

no contaban con una gran complejidad. Asimismo, el bloque de mayor complejidad, el controlador, ya se había verificado tanto por simulaciones como en implementaciones en placa. Por otro lado, cabe mencionar que simular el contenido de 10 bloques de memoria (5120 bytes, es decir 40960 bits) en un banco de pruebas, sin contar con un emulador de memoria, es una tarea compleja. Por estos motivos se optó por pasar directamente a la implementación en placa.

3.3.3. Implementación del host para la lectura de 10 bloques consecutivos

Siguiendo las instrucciones descritas en la sección 3.1 se guardó en la memoria SD un archivo binario con 10 bloques iguales. Este mismo bloque se cargó en la memoria **constant.array** del bloque host en Vivado. A continuación, se fueron colocando números de bytes aleatorios (de los 512) a verificar de cada uno de los 10 bloques. Asimismo, en el octavo bloque se colocó un byte erróneo adrede, es decir, en lugar de comparar el byte X leído de la memoria SD, con el byte X de **constant.array** cargado en el host, se lo comparó con otro dato incorrecto.

Se implementaron los bloques descritos en las subsecciones 3.2.2 y 3.2.3 en la placa de desarrollo. Los LEDs de la Nexys 4 DDR representan los siguientes estados:

- LED0 : Error. Se enciende cuando se produce algún error en la lectura de la memoria y la secuencia de estados de la máquina de estados del controlador finaliza en el estado **REPORT_ERROR**.
- LED1 : Esperar solicitud de lectura de host. Se enciende cuando el estado de la máquina de estados del controlador de memoria es **WAIT_FOR_HOST_RW**. Este Led permanece encendido cuando se ha finalizado el proceso de lectura y el controlador aguarda a que el host indique si se debe realizar otra operación de lectura.
- LED2 : Indicador de lectura **rd.i**. Permanece encendido durante el proceso de lectura y se apaga al finalizarlo.
- LED3 a LED5: Dos bytes de CRC. Se encienden al recibirse los bytes de CRC recibidos al final de un bloque de bytes.
- LED6 a LED15: Cada uno de estos 10 leds representa un byte de uno de los 10 bloques leídos de la memoria SD. Cuando los leds se encienden significa que ha coincidido el byte leído con el byte cargado manualmente en el arreglo.

En la Figura 3.16 se presenta el resultado obtenido de la serie de pruebas del controlador y el host en la Nexys 4 DDR.

Se observa que a excepción de los leds 0, 2 y 13, los demás se encuentran encendidos. En el caso de los leds 0 y 2 la causa es que no se han producido errores en el proceso de lectura y que se han leído correctamente los 10 bloques, respectivamente. Por otro lado, el estado apagado del LED13 indica que el byte aleatorio leído del octavo bloque de memoria no coincide con el dato que se compara. Esto es correcto dado que, tal como se mencionó anteriormente, se colocó

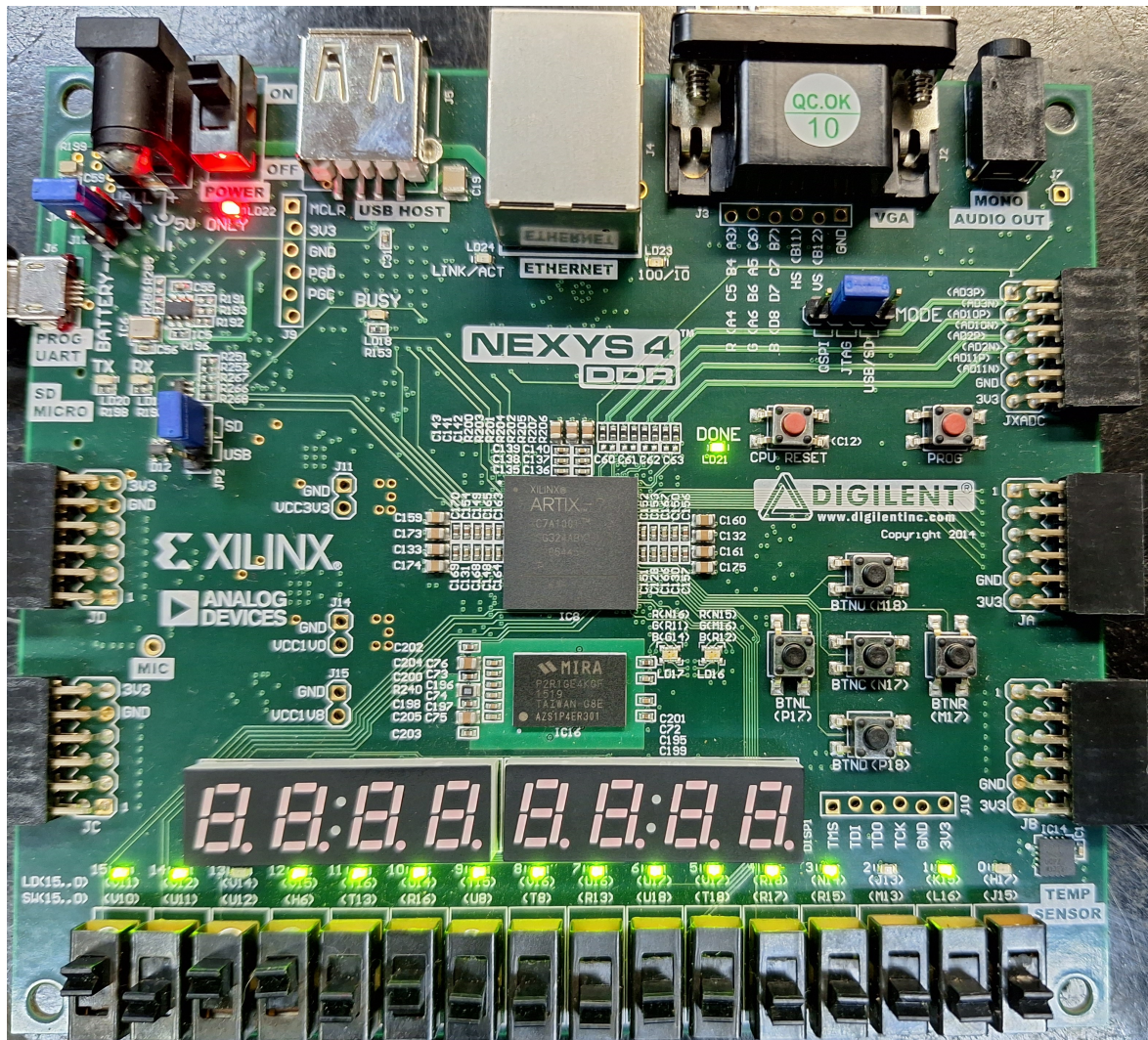


Figura 3.16: Resultado de prueba de implementación en placa.
Fuente: Elaboración propia.

un dato erróneo en este bloque para verificar el funcionamiento del host tanto para casos de coincidencia de datos como desigualdad de los mismos.

De esta manera se pudo verificar el correcto funcionamiento del controlador de memoria a partir de un diseño de un host de funcionalidad no necesariamente vinculada a la aplicación particular, pero de sencilla implementación. Asimismo, el host de prueba resultó de gran ayuda para la depuración del código, dado que los leds proveen de ayuda visual para mejorar el entendimiento de lo que está sucediendo a nivel interno en la FPGA.

4. Controlador del módulo PmodDA2

Para la transmisión de una señal analógica a partir de una digital se requiere de un módulo conversor DAC, que en el proyecto actual es el PmodDA2, presentado en la sección 2.5. Para la utilización de este módulo en la FPGA, se debe diseñar un controlador que asegure que los datos a convertir tengan el formato requerido por el conversor, y que sincronice la transferencia de estos datos. En el presente capítulo se realiza el diseño del controlador del módulo PmodDA2, y se verifica su correcto funcionamiento por medio de simulaciones e implementaciones en placa.

4.1. Formato de los datos de entrada

El módulo conversor PmodDA2 en su modo normal de operación admite palabras de 12 bits de datos precedidas por 1 nibble de ceros (ver subsección 2.5.2). Por comodidad de formato de datos, en la sección 3.1 se optó por extender con 4 ceros adicionales las palabras de 12 bits para facilitar el proceso de lectura. Como consecuencia, los datos almacenados en la memoria ya cuentan con un formato compatible con el conversor PmodDA2.

4.2. Diseño del controlador del módulo PmodDA2

Para que el DAC pueda convertir correctamente los datos deseados se requiere de un controlador que cumpla con las siguientes funcionalidades:

- Generar una señal de reloj **SCLK** de 25 MHz (menor a la frecuencia máxima de SCLK indicada en la Tabla 2.3).
- Generar una señal de sincronismo **SYNC** que habilite el proceso de conversión en el DAC.
- Generar una señal de aviso **DONE** cuando pueda ser realizada la carga paralela del dato a convertir.
- Realizar una carga paralela de los datos a convertir, dado que los datos leídos de la memoria SD se transmiten a través de un bus de 16 bits de ancho.
- Realizar una transmisión serial de los datos a convertir en el formato adecuado.

Teniendo en cuenta los items mencionados se diseñó un bloque controlador del PmodDA2 con 5 procesos que conforman una máquina de estados. En la Figura 4.1 se presenta un esquema simplificado del controlador diseñado con los 5 procesos que cumplen las funciones de los items descritos. Al igual que en el capítulo

anterior, los contornos continuos representan bloques mientras que los contornos discontinuos representan procesos.

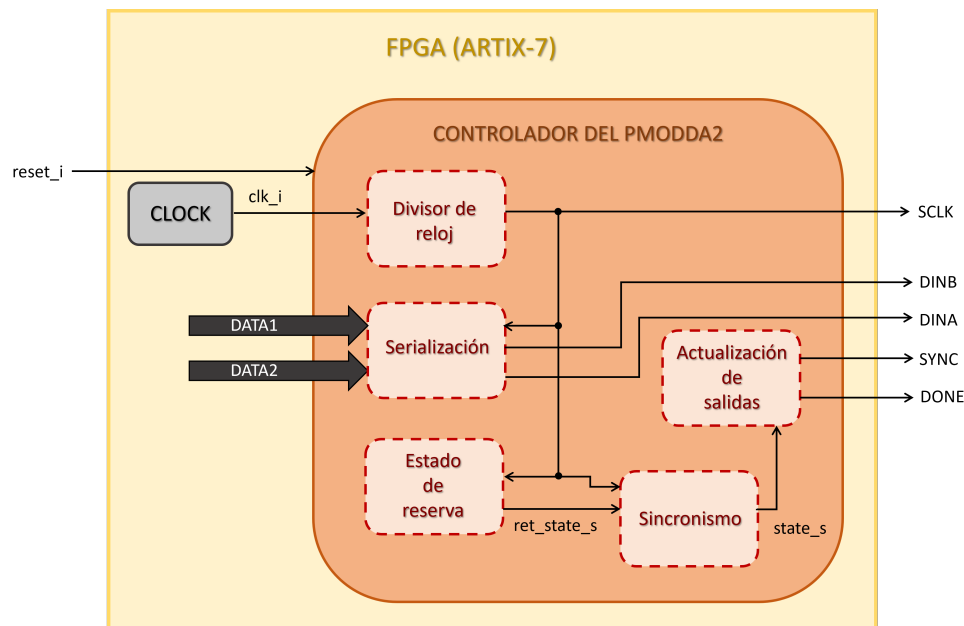


Figura 4.1: Esquema del controlador del PmodDA2.

Fuente: Elaboración propia.

La descripción de los pines de salida del controlador se realizó en la sección 2.5. Por otro lado, las entradas del bloque son: **reset_i**, **clk_i**, **DATA1**, **DATA2** y **START**. Las entradas **DATA1** y **DATA2** son buses de datos de 16 bits donde se realiza la carga paralela de los datos a enviar serialmente al conversor. Por otro lado, la entrada **START** controla el comienzo de serialización de las entradas paralelas (cuando se encuentra en un estado bajo) y controla el regreso al estado inactivo (**IDLE_STATE**) desde el estado **SYNC_DATA_STATE**.

A continuación se describen las funciones de cada uno de los procesos presentes el controlador del DAC.

El *proceso divisor de reloj* tal como lo indica su nombre, consiste en un contador (de 0 a 3) para la generación de una frecuencia de reloj de 25 MHz a partir del reloj principal de la FPGA de 100 MHz. Cuando el contador llega a 3, conmuta el valor de su salida de reloj, generando así un reloj de una frecuencia cuatro veces menor que la frecuencia del reloj que incrementa el contador.

El *proceso Serialización* es fundamental para el controlador del PmodDA2 puesto que se encarga de recibir los datos en formato paralelo y enviarlos serialmente al conversor. El diagrama de flujo de este proceso se presenta en la Figura 4.2. Si la señal de habilitación de carga paralela (**enParallelLoad_s**) se encuentra en alto entonces se cargan las entradas **DATA1** y **DATA2** a las señales auxiliares (copias para la serialización) **temp1_s** y **temp2_s**. Para que la señal de habilitación de carga paralela se encuentre en el estado alto, la entrada **START** también debe estar en alto. De esta manera, cuando la lógica de apliel controlador del DAC queda habilitada para cargar la entrada paralela. Posteriormente, el

proceso Serialización aguarda a la habilitación de desplazamiento de la señal almacenada (**enShiftCounter_s** = '1') que está determinada por el cambio de valor de la entrada **START** de alto a bajo. Para serializar las entradas se toman las señales auxiliares (**temp1_s** y **temp2_s**) y se las va desplazando hacia la derecha bit a bit. El bit desplazado se va colocando en las salidas serie (**DINA** y **DINB**) y se envía al convertor. Esta acción se realiza hasta haber transmitido los 16 bits de la palabra a convertir.

Si bien el *proceso Serialización* se diseñó para poder realizar la conversión de dos entradas paralelas a serie (dado que el módulo PModDA2 cuenta con dos convertidores), en este proyecto únicamente se utiliza una de las dos. De esta manera, el bloque diseñado puede ser implementado para futuras aplicaciones que sí requieran dos conversiones simultáneas.

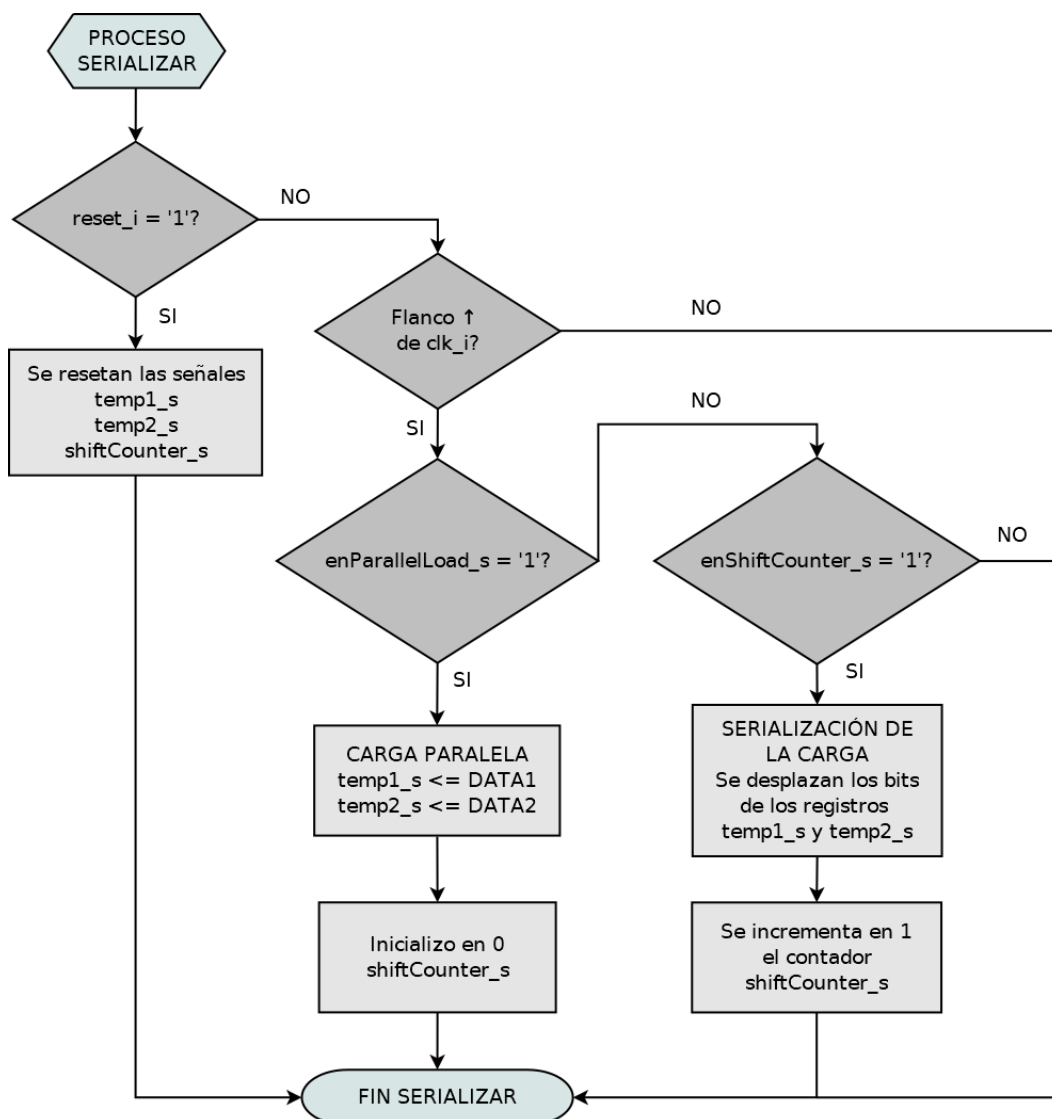


Figura 4.2: Proceso Serialización.
Fuente: Elaboración propia.

El *proceso Actualización de salidas* (Figura 4.3) es una máquina de estados que actualiza el valor de distintas salidas y señales internas en función del estado

actual. Las salidas que actualiza son **DONE** y **SYNC**. La salida **DONE** en estado alto le notifica al host que ha finalizado la serialización de la entrada y que el controlador puede recibir un dato nuevo. Por otro lado, tal como se mencionó anteriormente, la salida **SYNC** (en estado bajo) habilita al DAC para recibir los bits a convertir. Las señales internas actualizadas en este proceso son **enParallelLoad_s** y **enShiftCounter_s** que habilitan la carga paralela y el desplazamiento del registro contenedor de la palabra a serializar.

En el estado **Idle_state** se habilita la carga paralela y se le avisa al host que puede enviar una palabra por el bus de 16 bits. En el estado **shiftOut_state** se habilita la serialización de la entrada paralela y por último, en el estado **syncData_state** se envía el pulso de sincronismo al conversor.

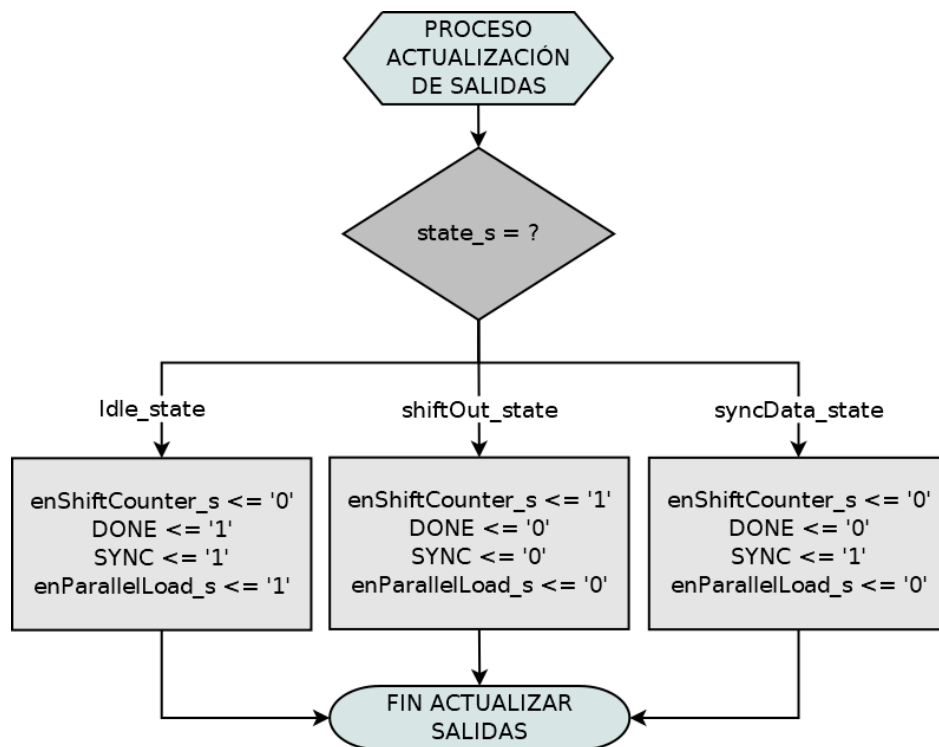


Figura 4.3: Proceso Actualización de salidas.

Fuente: Elaboración propia.

El *proceso Reserva* se encarga de actualizar el estado de reserva (**ret.state.s**) en función del valor de las entradas y señales mencionadas. Si se produce un cambio en la entrada **START** o la señal **shiftOut_state** entonces, el estado reservado es distinto del estado actual. En cambio, si los valores se mantienen, el estado reservado se mantiene en el estado actual. El diagrama de flujo del proceso se presenta en la Figura 4.4.

Por último, el *proceso Sincronismo* actualiza el valor del estado actual al valor del estado reservado (**ret.state.s**).

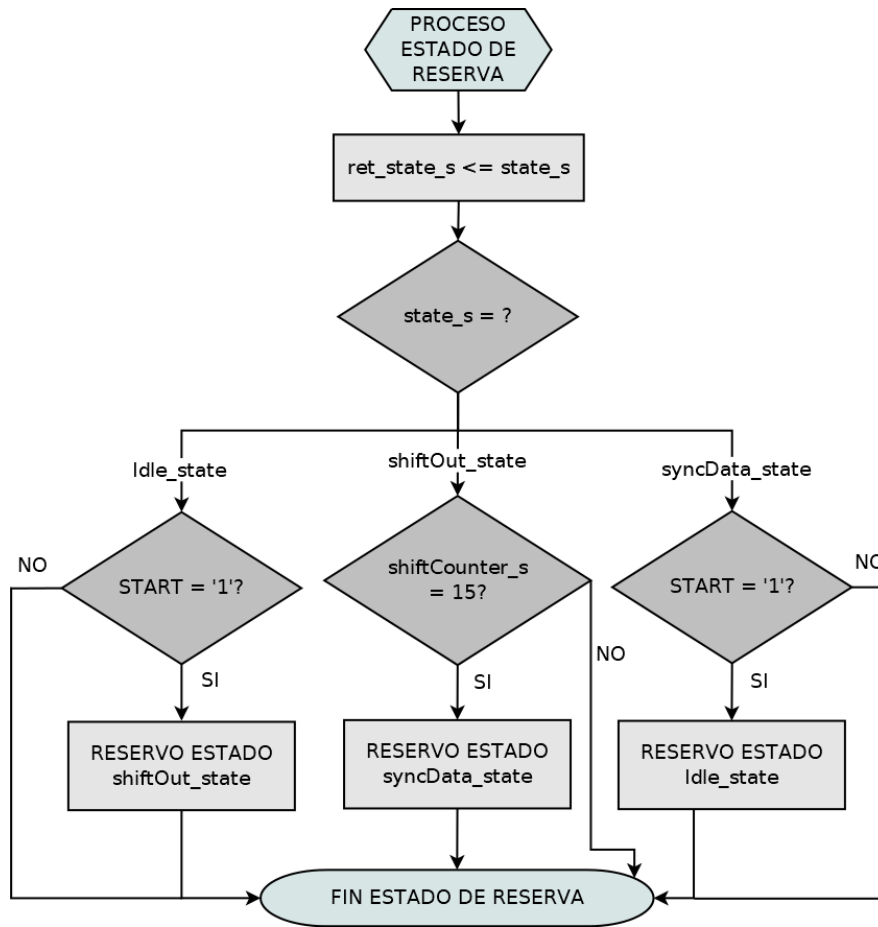


Figura 4.4: Proceso Estado de Reserva.
Fuente: Elaboración propia.

A partir de los procesos anteriores se obtiene una máquina de estados de tipo Moore, puesto que las salidas dependen únicamente del estado actual. Asimismo, cabe destacar que el cambio de estados se produce con el cambio de los valores de la entrada **START** y la señal **enShiftCounter_s**. Cada vez que la máquina de estados inicia en el estado **Idle_state** y regresa a él, se ha completado la conversión de una palabra digital a su correspondiente valor analógico en el DAC. En la Figura 4.5 se presenta la máquina de estados mencionada.

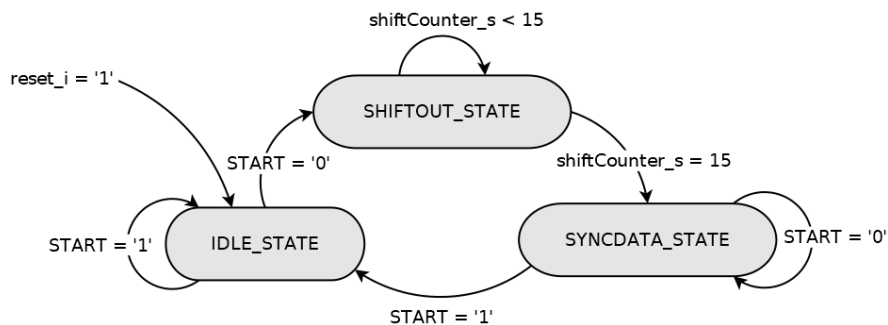


Figura 4.5: Máquina de estados principal del controlador del PmodDA2.
Fuente: Elaboración propia.

4.3. Simulación funcional del controlador del PmodDA2

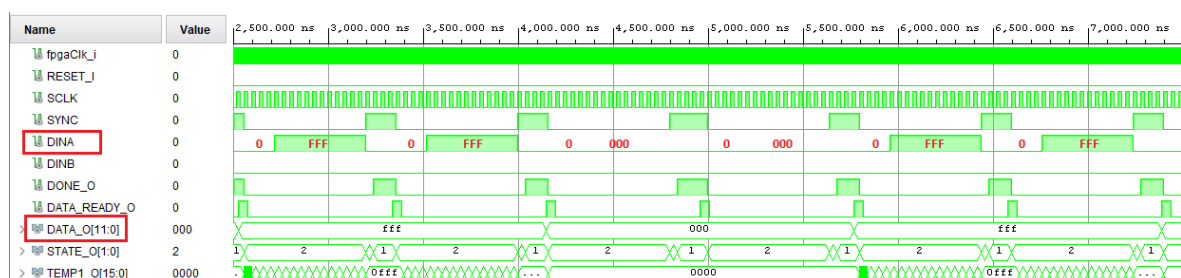
El controlador diseñado de la Figura 4.1 es un componente genérico, que podría estar conectado a cualquier elemento que envíe datos paralelos, en el formato adecuado. Por lo tanto, para verificar el comportamiento funcional del controlador diseñado, se añadió un bloque **sendData** que envía datos en formato paralelo para ser posteriormente convertidos.

El bloque **sendData** cuenta con una memoria de datos de 16 bits **constant_array_16** (similar a **constant_array** del código 3.5) a ser enviados al controlador. Asimismo, incluye un proceso que verifica que el controlador esté habilitado para recibir datos (**DONE** en alto) y cuando lo está, envía uno de los datos de la memoria y da aviso de esto colocando un estado alto en la entrada **START**. A medida que se envían los datos, se actualiza la posición de memoria **constant_array_16** a leer y cuando se ha leído la totalidad de esta memoria, se resetea la dirección de lectura a cero. De esta manera, se envían constantemente datos al controlador para ser serializados.

En la Figura 4.6 se presentan los resultados de la simulación del controlador del PmodDA2 junto con el bloque **sendData**. En **constant_array_16** se cargaron 2 ceros consecutivos h0000, seguidos de dos caracteres h0FFF, que corresponden al máximo nivel de tensión de salida del DAC.

En la Figura 4.6(a) se observa que la entrada paralela **DATA_O** se serializa correctamente en la salida **DINA**. Asimismo se observa que el controlador solo le envía datos al conversor digital-analógico cuando la salida **SYNC** se encuentra en bajo.

Por último, en la Figura 4.6(b) se hace hincapié en la presencia de los 16 pulsos del reloj **SCLK** de 25 MHz, que existen entre los pulsos de la señal **SYNC**. Los datos serializados se cargan en el DAC con los flancos ascendentes de la señal de reloj **SCLK** por lo cual el tiempo de muestreo es de 1,25 MSa/s (a los 16 bits de datos se le suman los 4 pulsos de reloj en los que **SYNC** se encuentra en alto).



(a) Serialización de las entradas paralelas.


```
13  40 => x"46E", 41 => x"42F", 42 => x"3F0", 43 => x"3B1",
14  44 => x"372", 45 => x"333", 46 => x"2F4", 47 => x"2B5",
15  48 => x"276", 49 => x"237", 50 => x"1F8", 51 => x"1B9",
16  52 => x"17A", 53 => x"13B", 54 => x"0FC", 55 => x"0BD",
17  56 => x"07E", 57 => x"03F", 58 => x"000"
18 );
```

Para la realización de la prueba se conectó el módulo PmodDA2 a la placa Nexys4 y se utilizó un osciloscopio para la medición de la señal analógica convertida. En la Figura 4.7 se presenta la disposición del módulo PmodDA2 con la placa de desarrollo y el osciloscopio para las pruebas experimentales.

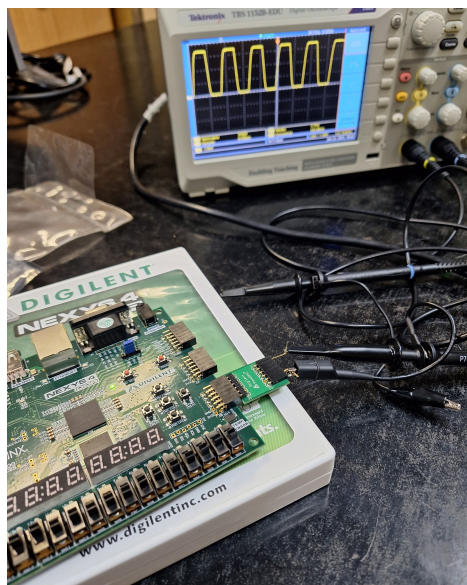


Figura 4.7: Disposición de los instrumentos utilizados para las pruebas experimentales.

Fuente: Elaboración propia.

En la Figura 4.8 se presenta la forma de onda graficada a la salida del PmodDA2. Se observa que un período de la onda graficada es aproximadamente de $50 \mu\text{s}$ y su amplitud de 1,5 V, tal como se había anticipado.

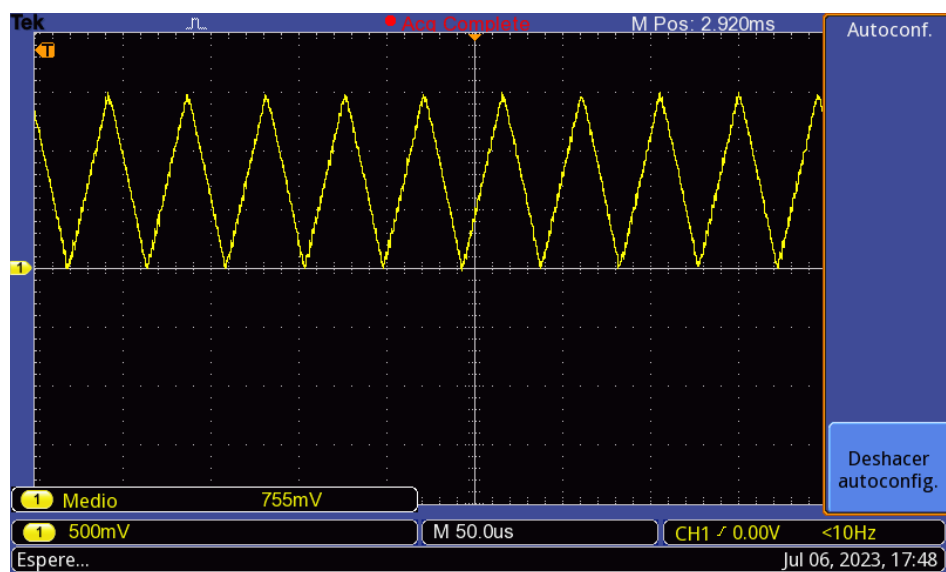


Figura 4.8: Visualización en el osciloscopio de una onda triangular para la verificación del controlador del PmodDA2.

Fuente: Elaboración propia.

5. Interfaz entre el controlador y el DAC

Una vez completado el diseño del controlador de la memoria SD y del PmodDA2, se comenzó con el diseño de la interfaz entre estos bloques y la adición de los componentes finales para el funcionamiento del Emisor. En primera instancia se diseñó una interfaz directa, de forma tal que el DAC recibiera y convirtiera los datos en la medida en que estos eran leídos de la memoria SD. La interfaz directa presentó inconvenientes, que se mencionarán a lo largo del capítulo, por lo cual se diseñó una interfaz con una memoria RAM intermedia. Esta interfaz, que almacena los datos leídos en una etapa intermedia antes de enviarlos al DAC para su conversión, es la que se utiliza en el diseño final del emisor.

Luego de las descripciones de las interfaces, se presentan las últimas incorporaciones al emisor, que permiten la selección de la señal a ser transmitida y el sincronismo del módulo con otros emisores. Por último, se presentan mediciones de distintas señales codificadas y moduladas a la salida del módulo PmodDA2.

5.1. Interfaz directa

5.1.1. Esquema de la interfaz directa entre el controlador de memoria y el Pmod

Para la representación analógica de las señales digitales leídas de la memoria SD, se propuso en primera instancia una interfaz directa entre los bloques de los capítulos 3 y 4. Para ello se introdujeron algunas modificaciones en el bloque host diseñado en la sección 3.3.1 y se introdujeron bloques adicionales que garanticen el sincronismo entre los datos leídos por el controlador y los convertidos por el PmodDA2. En la Figura 5.1 se presenta el esquema diseñado¹. Los bloques del controlador de memoria, controlador del PmodDA2 y de lectura multibloque son los que se presentaron anteriormente y no cuentan con modificaciones.

El bloque *Host Control* incluye el *proceso Lectura* del host presentado en la subsección 3.3.1 (Figura 3.13) con algunos cambios. Además de realizar el handshake con el controlador, se le añadió la función de concatenar los bytes leídos de a pares. De esta manera, se obtienen las palabras de 16 bits a ser convertidas. Adicionalmente, cuando la palabra se encuentra disponible, se coloca un valor alto en la señal **word_ready**. Posteriormente se detecta el flanco ascendente de esta

¹Dado que los componentes de la interfaz interactúan con componentes o procesos que pertenecen al bloque *Host*, se optó por incluirlos también dentro de este bloque, en lugar de generar un bloque nuevo denominado Interfaz. Adicionalmente, es la Interfaz la que indica cuando se puede realizar otra lectura de la memoria SD, por lo cual cumple funciones de Host de memoria SD.

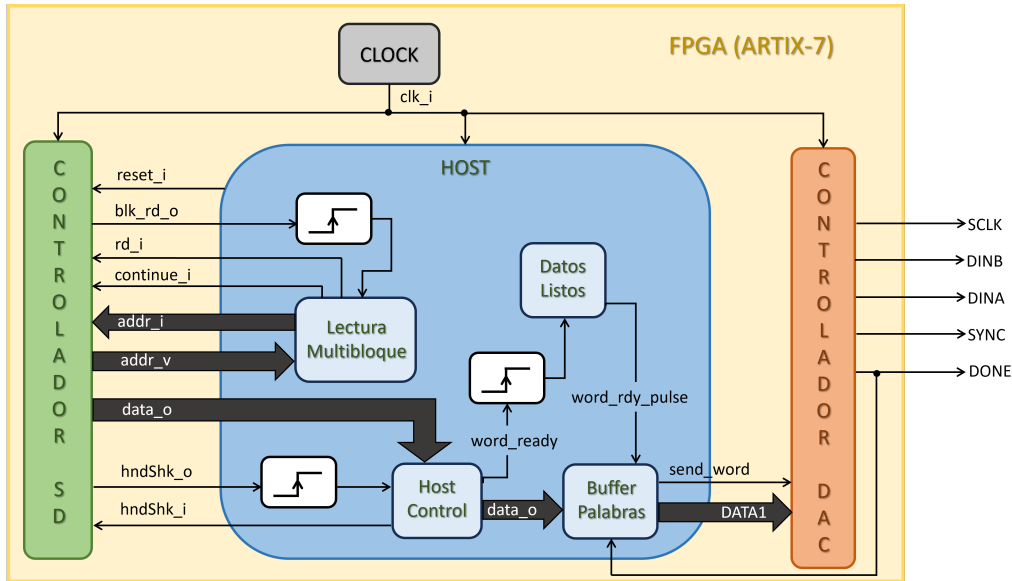


Figura 5.1: Esquema de la interfaz directa entre el controlador de memoria SD y el PmodDA2.

Fuente: Elaboración Propia.

señal (mediante el bloque de *detección de flanco de reloj*) y este flanco se utiliza para generar un pulso (**word_rdy_pulse**) de 10 períodos de reloj, en el bloque *Datos Listos*. Por último, las palabras del bloque *Host Control* ingresan a un *Buffer de palabras* cuando *word_rdy_pulse* se encuentra en un valor alto. Este bloque tiene adicionalmente como entrada la salida **DONE** del bloque de interfaz con el PmodDA2. Por lo cual el *Buffer de Palabras* carga la palabra leída de la memoria cuando ésta se encuentra disponible, pero solo la envía a la interfaz con el DAC cuando éste ha finalizado la última conversión. De este modo, se garantiza que todos los datos leídos sean convertidos.

5.1.2. Simulación de la interfaz directa

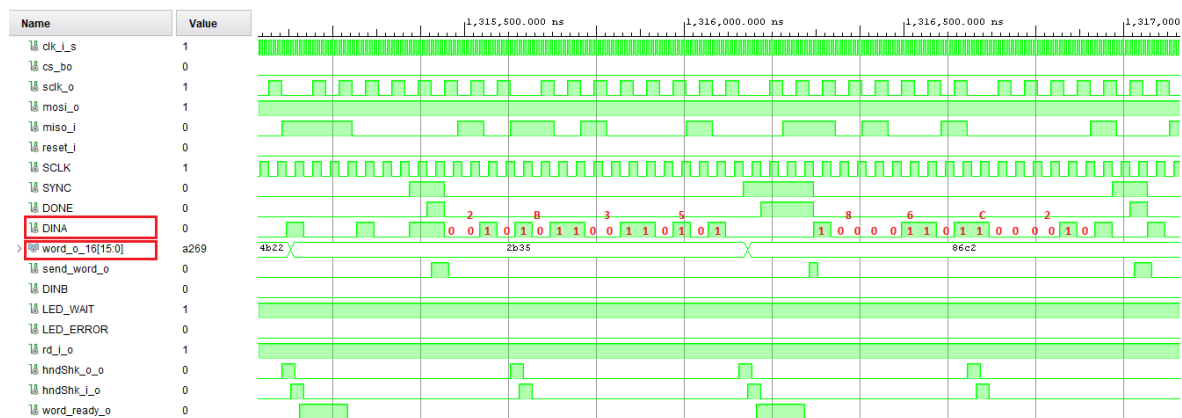
Al igual que en los capítulos anteriores se realizó una simulación de los bloques diseñados, emulando la lectura de datos de la memoria SD por medio de la generación de una trama de bits aleatorios en el puerto **miso.o**.

En la Figura 5.2 se presentan los resultados de 2 simulaciones realizadas para verificar el diseño de la interfaz. En la Figura 5.2(a) se observa que las palabras a la salida del bloque *Host Control* (**word.o_16**) son correctamente serializadas a la salida del controlador del PmodDA2 (**DINA**), validando su funcionamiento. No obstante, en la Figura 5.2(b), donde se presenta la simulación de dos bloques de memoria consecutivos, se observa un problema que presentaría la interfaz diseñada. Alrededor de los 1350 μ s de simulación, las señales de handshake entre el controlador y el host permanecen en cero durante varios períodos de reloj. Esto sucede porque al finalizar la lectura de un bloque se realizan distintas acciones, como actualizar la dirección del bloque a leer y recibir el token de inicio de datos del siguiente bloque, entre otras. Por lo tanto, existe un breve intervalo en el cual

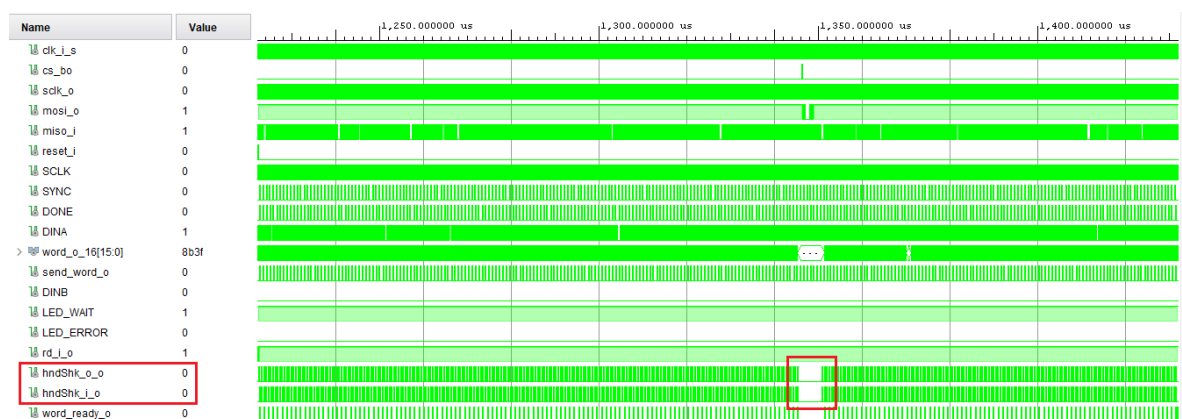
no se leen datos de la memoria.

Si la señal a ser transmitida ocupa más de un bloque de memoria (como ocurre con todas las señales codificadas y moduladas del proyecto) no se obtiene una representación correcta de la misma. En cambio, se transmiten correctamente 256 palabras convertidas a sus correspondientes valores analógicos y luego la salida es nula durante algunos ciclos de reloj. Posteriormente se repite esta situación.

Puesto que la velocidad de lectura del controlador de memoria SD es menor a 1 MSa/s (en la Figura 3.10 se tienen 16 bits en $1,7 \mu s$ por lo cual es de alrededor 0,6 MSa/s), tampoco es posible acelerar la lectura de datos para compensar la conversión de muestras en los intervalos donde no se lee la memoria SD. Como consecuencia, se descartó la interfaz directa entre el controlador y el conversor y se optó por introducir una memoria RAM auxiliar entre el controlador de la memoria SD y el controlador del PmodDA2.



(a) Simulación de un bloque de memoria.



(b) Simulación de más de un bloque de memoria, donde se indica el instante donde no se transmiten datos por el DAC.

Figura 5.2: Simulación de la interfaz directa entre el controlador de memoria y el PmodDA2.

Fuente: Elaboración propia.

5.1.3. Implementación de la interfaz directa

A partir de una implementación en placa se verificó, mediante mediciones realizadas a la salida del DAC con el osciloscopio, que cada 512 bytes el módulo PmodDA2 no recibe datos para ser convertidos. En la señal amarilla de la Figura 5.3 se observan los intervalos en los que la salida **DINA** toma valores nulos, mientras que la señal celeste es la **SYNC**. De esta manera, se pudo verificar el mismo resultado que en la simulación capturada en la Figura 5.2.

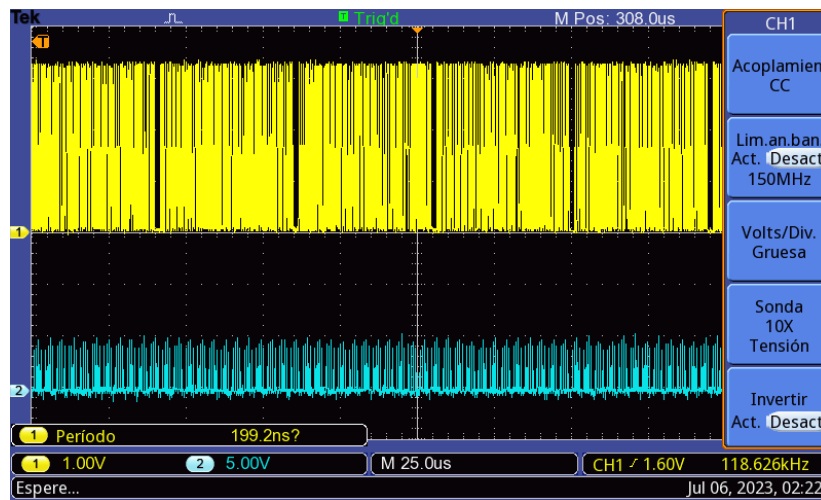


Figura 5.3: Captura del osciloscopio con la implementación de la interfaz directa.
Fuente: Elaboración propia.

5.2. Interfaz con memoria

Teniendo en cuenta las dificultades mencionadas del diseño de la interfaz directa, se añadió una etapa de almacenamiento intermedia entre el controlador de memoria y el DAC. Luego de realizar un reseteo mediante la entrada **reset.i**, la memoria SD es leída y sus datos se vuelcan en una memoria RAM auxiliar de la FPGA. Finalizada la lectura de la memoria SD, comienza la transmisión de los datos, por medio de la lectura de la memoria auxiliar. Por lo tanto, la memoria SD se lee una única vez y posteriormente se trabaja con los datos almacenados en la memoria auxiliar. Como esta última no está fraccionada en bloques de 512 bytes, los datos pueden ser leídos y transmitidos sin presentar intervalos nulos en la salida del transmisor.

5.2.1. Esquema de la interfaz con la memoria RAM auxiliar

En la Figura 5.4 se presenta el sistema con la interfaz a diseñar. Los bloques del controlador, de lectura multibloque, el Host Control y la interfaz con el PmodDA2, son los mismos que se presentaron anteriormente.

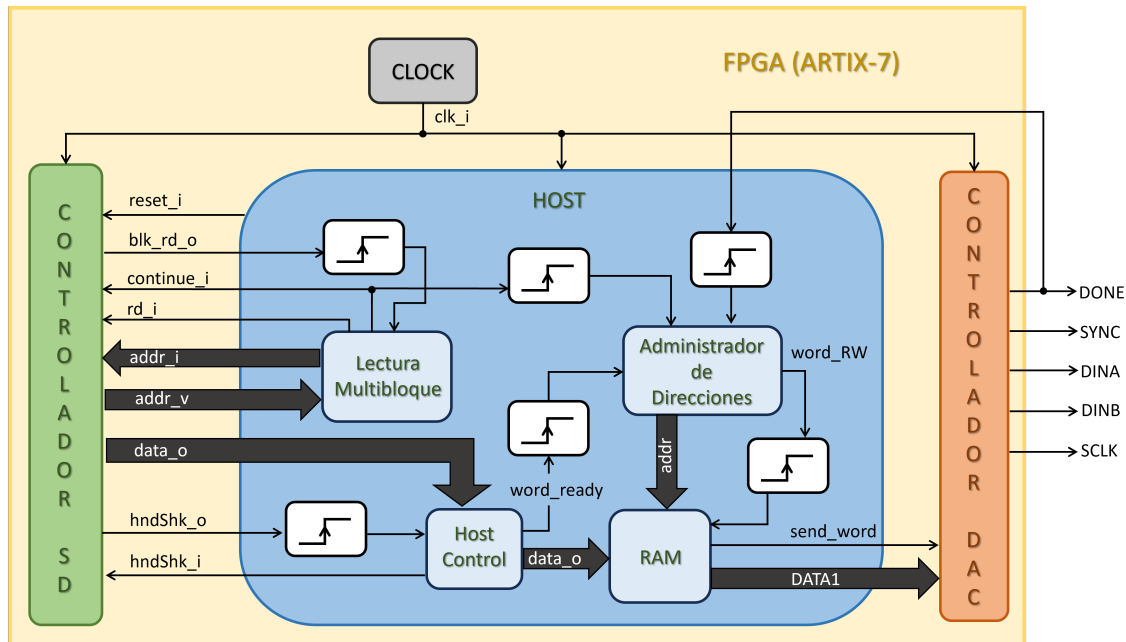


Figura 5.4: Esquema de la interfaz con la memoria auxiliar.
Fuente: Elaboración propia.

Se incorporan el bloque *Administrador de direcciones* y el bloque *memoria RAM*. Estos bloques funcionan coordinadamente para asegurar que todos los datos de la memoria SD sean leídos y almacenados en la memoria RAM, y a su vez que estos sean transmitidos por el DAC.

Teniendo en cuenta que la longitud de los vectores de señales codificadas y moduladas no supera los 40 kWords, se diseñó una *memoria RAM* de 200 kWords. Puesto que se decidió que esta memoria solo almacenaría la señal a ser transmitida (en lugar de las 4 señales posibles), se observa que se cuenta con espacio de sobra, ante un eventual crecimiento en la cantidad de muestras. El bus de direcciones para una memoria del tamaño escogido tiene un ancho de 18 bits.

Cabe aclarar, que si la cantidad de muestras a transmitir superara las $200 \cdot 2^{10}$ muestras, se puede modificar fácilmente el tamaño de la memoria RAM para adecuarse a los requisitos de la aplicación (siempre y cuando no se superen las limitaciones de recursos de la FPGA). Esto es posible gracias a que las características de la memoria se fijaron mediante constantes, tal como puede observarse en el extracto de Código 5.1.

Código 5.1: Características de la RAM de 200 kWords fácilmente editables.

```

1  entity RAM_200k is
2      generic (
3          RAM_DATA_WIDTH    : positive := 16;
4          RAM_ADDR_WIDTH    : positive := 18;
5      );
6  
```

```

7  -- Resto de la entidad
8
9  architecture arch of RAM_200k is
10
11  constant SIZE_KB    : integer := 200;
12  constant RAM_DEPTH : integer := SIZE_KB*1024;
13
14  -- Resto de la arquitectura

```

La RAM_200k cuenta con una entrada de habilitación activada por un flanco ascendente de la señal **word_RW** y una entrada de habilitación de lectura o escritura **rd_i**. Asimismo, posee una salida (**send_word**) que se conecta con la entrada **START** del controlador del PmodDA2. El bloque *Administrador de direcciones* accede a la RAM_200k por medio de la dirección indicada en su bus de salida **addr** y actuando sobre su entrada de habilitación **word_RW**.

En la Figura 5.5 se presenta el diagrama de flujo del bloque *Administrador de direcciones*. La operación a realizarse (lectura o escritura) está determinada por el valor de la entrada **rd_i** proveniente del bloque *lectura multibloque*. Cuando **rd_i** está en alto, se están realizando operaciones de lectura de la memoria SD. Por lo tanto, si se recibe un flanco ascendente de **word_ready** (proveniente del bloque *Host_Control*) se incrementa la dirección de memoria a escribir y se coloca un alto en la salida **word_RW**.

Por otro lado, si **rd_i** se encuentra en bajo, ha finalizado el proceso de lectura. Como consecuencia, los datos almacenados en la memoria RAM_200k ya pueden comenzar a ser transmitidos. En el momento en el que **rd_i** pasa de alto a bajo se resetea el valor de **addr** para comenzar a leer la memoria RAM_200k desde el principio. Si se recibe un flanco ascendente de **DONE**, el PmodDA2 ya finalizó la última conversión, por lo que se incrementa **addr**, se coloca un 1 en la salida **word_RW** y se resetea la señal **counter**. Si no se tiene ni un flanco descendente de **rd_i** ni un flanco ascendente de **DONE** con **rd_i** en bajo, se verifica el valor de la señal **counter**. En caso de que sea distinta de 2, se incrementa su valor.

Finalmente, si no se cumple ninguna de las condiciones previamente mencionadas se coloca un cero en la salida **word_RW** y se resetea la señal **counter**. Se incluye el contador en este bloque para garantizar que el pulso **word_RW** tenga la duración suficiente para ser detectado por el bloque de memoria RAM.

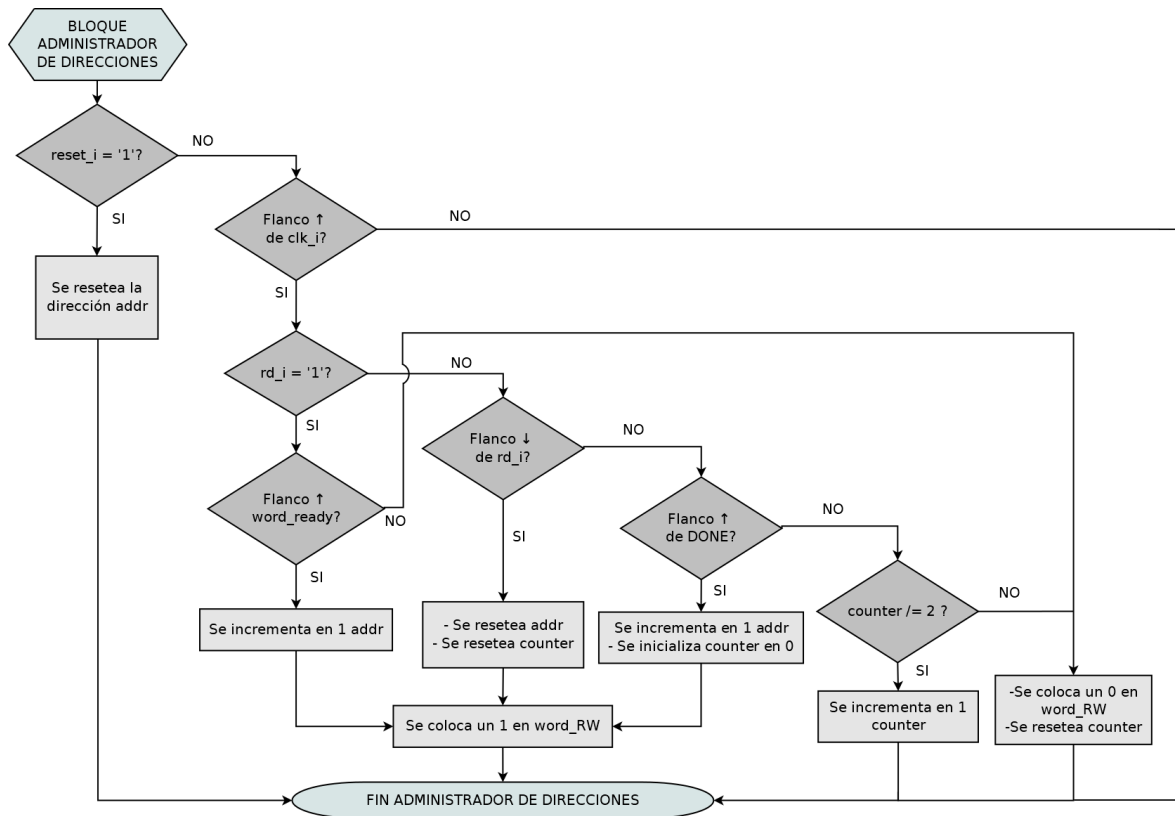
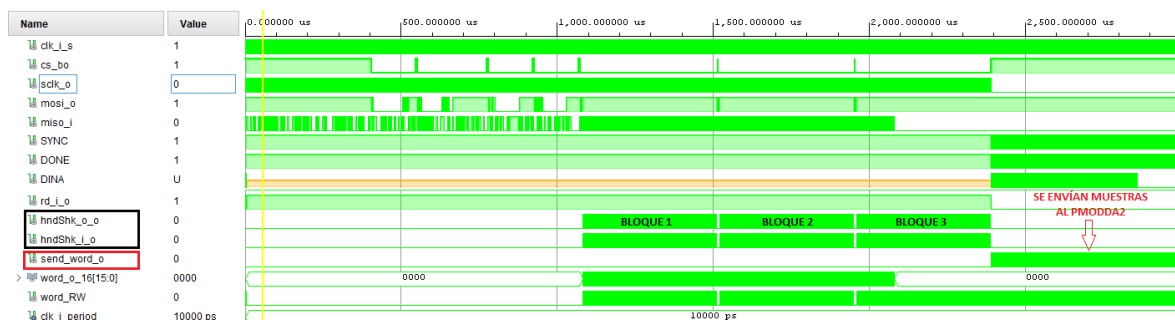


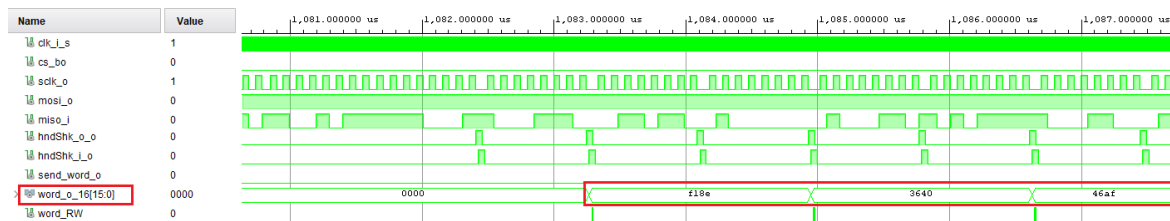
Figura 5.5: Diagrama de flujo del bloque Administrador de Direcciones.
Fuente: Elaboración propia.

5.2.2. Simulación de la interfaz con memoria auxiliar

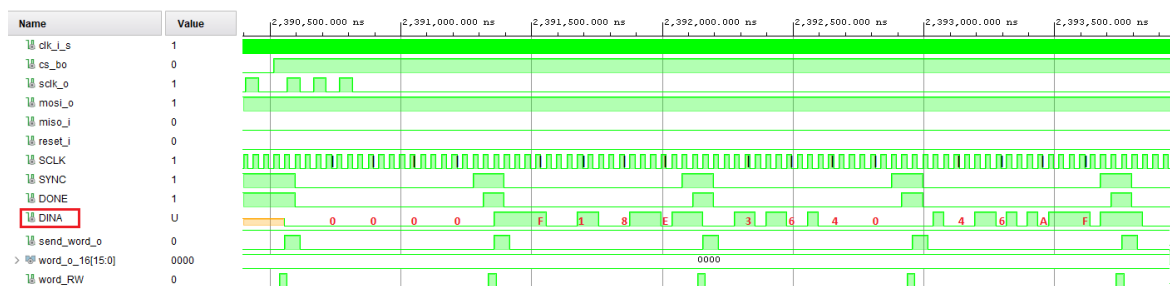
En la Figura 5.6 se presentan los resultados de la simulación realizada para 3 bloques de datos. En la Figura 5.6(a) se observa la simulación completa donde se puede distinguir que primero se realiza la lectura de 3 bloques ficticios de la memoria SD y al finalizar se serializan los datos almacenados en la memoria auxiliar. Por otro lado, en las Figuras 5.6(b) y 5.6(c) se verifica que los primeros datos leídos de la memoria SD (o su emulación en el banco de pruebas) coinciden con los primeros datos serializados. Por lo tanto el diseño funciona correctamente.



(a) Simulación de los 3 bloques de 512 bytes.



(b) Palabras leídas de la memoria SD ficticia.



(c) Palabras a la salida de la interfaz con el PmodDA2.

Figura 5.6: Simulación del sistema con la interfaz con memoria auxiliar.
Fuente: Elaboración propia.

5.2.3. Implementación de la interfaz con memoria auxiliar

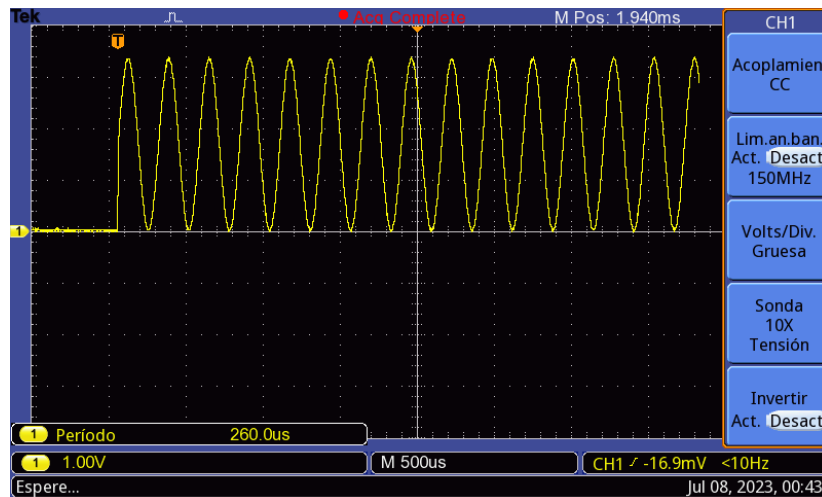
Para la verificación en placa del sistema diseñado, se almacenaron en la memoria SD las muestras de 20 períodos de una onda senoidal. En la Figura 5.7(a) se observa la salida analógica del PmodDA2 medida con el osciloscopio. Por otro lado, en la Figura 5.7(b) se presenta la repetición de los 20 períodos de señal senoidal en intervalos periódicos de tiempo. Esto se produce como consecuencia del overflow en el direccionamiento de la memoria RAM, por lo que **addr** vuelve a ser cero y se vuelve a representar la señal almacenada en la memoria RAM.

5.3. Incorporaciones finales al diseño

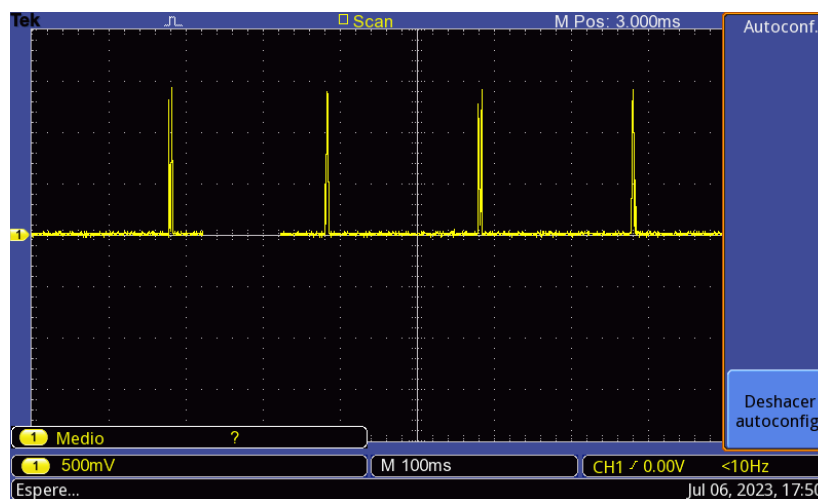
5.3.1. Ajuste de la frecuencia de muestreo

La frecuencia de muestreo obtenida en la sección 4.4 (1,19 MSa/s) no resulta de utilidad para la representación de señales analógicas de un período determinado. Si se quisiera obtener una señal analógica de 1 kHz a partir de muestras digitales con un DAC de 100 kSa/s, se necesitarían 100 muestras de cada período de la señal. Ahora bien, si la frecuencia de muestreo es 1,19 MSa/s y se quiere representar una señal de 100 kHz, se necesitarían 11,9 muestras por período, lo cual no es posible. Como consecuencia, se optó por ajustar la frecuencia de muestreo a 1 MSa/s.

Para la modificación en la frecuencia de muestreo se sustituyó el proceso *Divisor de reloj* por el componente primitivo administrador de reloj MMCME2_ADV



(a) Senoidal graficada.



(b) Periodicidad de las muestras graficadas por overflow de la salida addr.

Figura 5.7: Capturas del osciloscopio con la implementación de la interfaz con memoria RAM auxiliar.

Fuente: Elaboración propia.

(Advanced Mixed Mode Clock Manager). A partir de este pueden generarse señales de reloj con ciclo de trabajo del 50 % de un amplio rango de frecuencias.

En el Código 6.3 presentado en el Anexo se puede observar la instanciación del MMCME2_ADV para obtener el reloj **SCLK** de 20 MHz (antes de 25 MHz) a partir del de 100 MHz.

Además de realizar un ajuste de la frecuencia de **SCLK**, se disminuyó la duración del pulso de la señal **SYNC**, de 4 (ver sección 4.4) a 3. De esta manera, al tener 20 pulsos de **SCLK** por conversión se obtiene una frecuencia de muestreo de 1 MSa/s.

En la Figura 5.8 se presenta la señal de **SYNC** obtenida luego de la modificación del reloj que sincroniza al PmodDA2. Allí se observa que la frecuencia de la señal

de **SYNC** es de 1 MHz por lo que las muestras digitales se representan a un ritmo de 1 MSa/s.

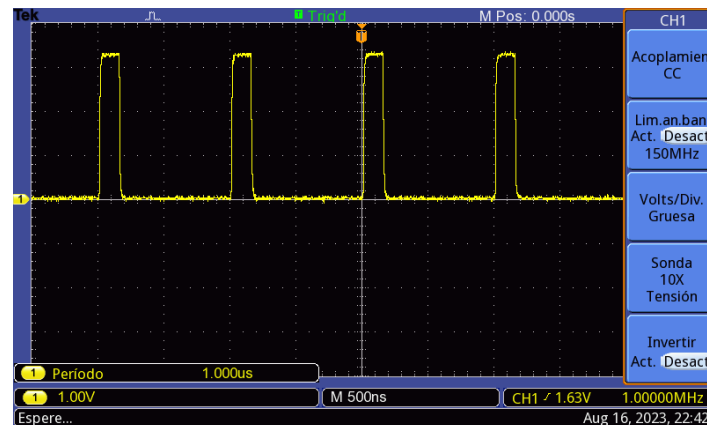


Figura 5.8: Captura del osciloscopio con la señal SYNC que controla la conversión de datos del PmodDA2

5.3.2. Selección de señal

En la sección 5.2 se obtuvo un sistema funcional para transmitir una señal almacenada en la memoria SD. Como el usuario debe poder elegir qué señal transmitir dentro de un determinado conjunto de señales codificadas y moduladas, se incorporó un bloque de *selección de señal* al sistema anterior que interactúa con el bloque *lectura multibloque*. El usuario selecciona la señal a transmitir por medio de interruptores que determinan la dirección inicial de memoria a leer.

Es importante recordar que según lo descrito en la sección 3.1, las 4 señales codificadas y moduladas se almacenan consecutivamente en la memoria SD. Por lo tanto, modificando la dirección inicial de lectura se puede acceder a las distintas señales almacenadas.

El bloque *selección de señal* modifica la dirección inicial y la envía al bloque *lectura multibloque*. Este último actualiza la última posición de lectura sumando el offset de tamaño de archivo. Cabe mencionar que al elegirse la señal que se desea transmitir, solo se leen las posiciones de memoria que contienen la señal correspondiente, es decir, que la memoria RAM auxiliar almacena una sola señal de las cuatro contenidas en la memoria SD.

En la Figura 5.9 se presentan los interruptores de la Nexys 4 a ser accionados para la transmisión de las señales.

Para escoger la señal a transmitir el usuario debe en primera instancia colocar el interruptor **reset.i** en su tope superior. Posteriormente activa el interruptor 1, 2, 3 o 4 en función de la señal que desea transmitir. El interruptor 1 selecciona la primera señal almacenada en la memoria SD, el interruptor 2 la segunda, y así sucesivamente. Una vez activado el interruptor de selección de señal se baja el interruptor de **reset.i** y en breve comienza a transmitirse la señal escogida.



Figura 5.9: Selección de la señal a transmitir por medio de interruptores en la Nexys 4 DDR.

Fuente: Elaboración propia.

En el extracto de código 5.2 se presenta el componente *signalSelect* encargado de configurar la dirección inicial de lectura de la memoria a partir de la posición de los interruptores de la Figura 5.9.

Código 5.2: Descripción VHDL del seteo de la dirección inicial de lectura por medio de los interruptores.

```

1  library IEEE,SDController;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use SDController.signalSelectPkg.all;
4
5  entity signalSelect is
6      Port ( clk_i : in STD_LOGIC;
7            SW : in STD_LOGIC_VECTOR (3 downto 0);
8            addr_init_o : out STD_LOGIC_VECTOR (31 downto 0));
9  end signalSelect;
10
11 architecture arch of signalSelect is
12
13 begin
14
15 process(clk_i,SW)
16 begin
17     if rising_edge(clk_i) then
18         case SW is
19             when x"01" => addr_init_o <= x"00008020";
20             when x"02" => addr_init_o <= x"000080B4";
21             when x"04" => addr_init_o <= x"00008148";
22             when x"08" => addr_init_o <= x"000081DC";
23             when others => null;

```

```

24         end case;
25     end if;
26 end process;
27
28 end architecture;

```

En el caso de que se levante más de un interruptor de selección de señal a la vez con **reset_i** también levantada, no se transmite ninguna señal. Por otro lado, si se accionara algún interruptor luego de la secuencia previa, no se produce ningún cambio en la señal transmitida, dado que el accionamiento de los interruptores solo produce una modificación de la dirección inicial de lectura cuando **reset_i** se encuentra en alto. Esto se puede observar el extracto de código 5.3 donde se presenta una porción de la descripción VHDL del componente *Lectura Multibloque*. Allí se aprecia que las direcciones iniciales y finales de lectura se cargan (a partir de la salida **addr_init_o** del componente *signalSelect*) cuando **reset_i** se encuentra en alto.

Código 5.3: Descripción VHDL del seteo de la dirección inicial de lectura en el componente Lectura Multibloque.

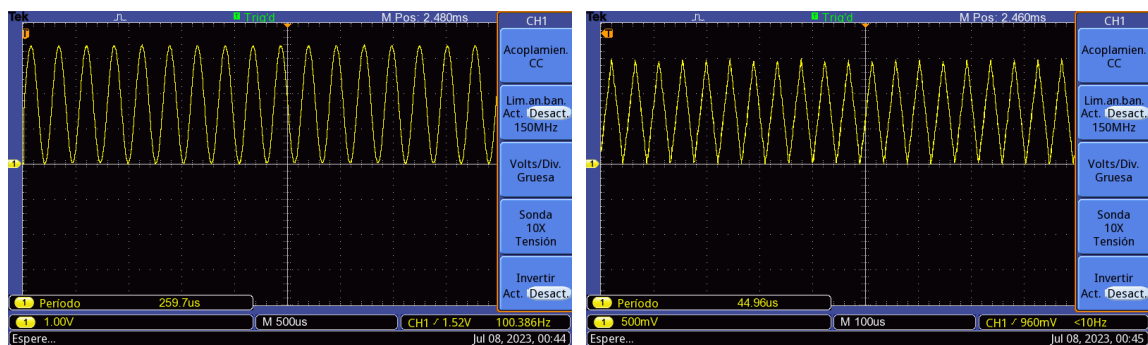
```

1  begin
2  if rising_edge(clk_i) then
3      if reset_i = '1' then
4          addr_save := to_integer(unsigned(addr_init_i)); -- addr_init_i
           → está conectada con addr_init_o del componente signalSelect
5          addr_i    <= addr_init_i;                      -- La
           → dirección de lectura se carga solo cuando reset_i = '1'
6          addr_fin  := addr_save + FILE_SIZE/512 - 1 ;
7          addr_fin_bin :=
           → std_logic_vector(to_unsigned(addr_fin,addr_fin_bin'length));
8
9          -- Resto de la descripción HDL del componente Lectura Multibloque
10         end if;
11     end if;

```

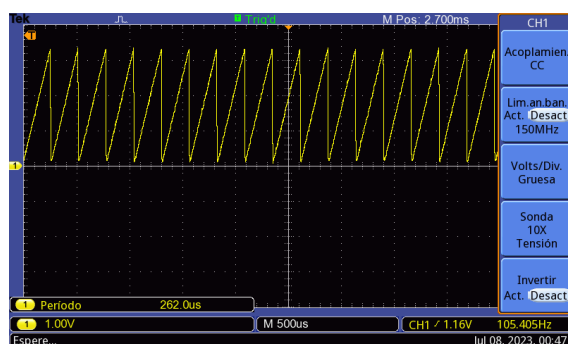
Implementación

Para la verificación en placa de la selección de señales se guardaron en la memoria SD 2 señales senoidales de distinta frecuencia, una señal triangular y una señal diente de sierra. En las Figura 5.10 se presentan las señales representadas a la salida del conversor, siendo estas escogidas por medio de los interruptores presentados anteriormente.



(a) Interruptor 1.

(b) Interruptor 2.



(c) Interruptor 3.

Figura 5.10: Capturas del osciloscopio de las señales elegidas por medio de los interruptores de la Nexys 4 DDR.

Fuente: Elaboración propia.

5.3.3. Disparo por estímulo externo

Teniendo en cuenta la aplicación particular para la cual se diseñó el transmisor, las señales moduladas y codificadas se enviarían a un período de repetición determinado por un estímulo externo. La finalidad es sincronizar un conjunto de múltiples emisores, donde cada uno de ellos cuente con una secuencia diferente de la misma familia de códigos. Para realizar una emisión simultanea, se sincronizan los emisores utilizando la señal PPS de un módulo GPS, emitida cada un segundo.

Por consiguiente, se introdujeron modificaciones al bloque *Administrador de direcciones* para reiniciar la lectura y transmisión de datos de la memoria auxiliar al recibirse un pulso positivo de una señal externa. Cuando se alcanza la máxima dirección de la memoria RAM auxiliar, el bloque no realiza ninguna operación hasta recibir un pulso en su entrada. Si se intentara disparar al emisor antes de que finalice la transmisión del contenido completo de la RAM auxiliar, no sucede nada: la transmisión de los datos tiene que haber finalizado para poder re-disparar el emisor.

Para verificar el funcionamiento de las modificaciones introducidas al bloque *Administrador de direcciones* se instanciaron todos los bloques de la sección anterior, en un módulo de mayor jerarquía al que se denominó *Emisor*. Además de las

entradas y salidas previamente mencionadas, se le añadió una entrada adicional por la que ingresarían los pulsos de retransmisión de señal. La entrada adicional PPS, ingresará eventualmente al emisor por alguno de los puertos Pmod (no utilizados) de la placa de desarrollo Nexys 4 DDR. En la Figura 5.11 se presenta el diagrama en bloques simplificado del Emisor.

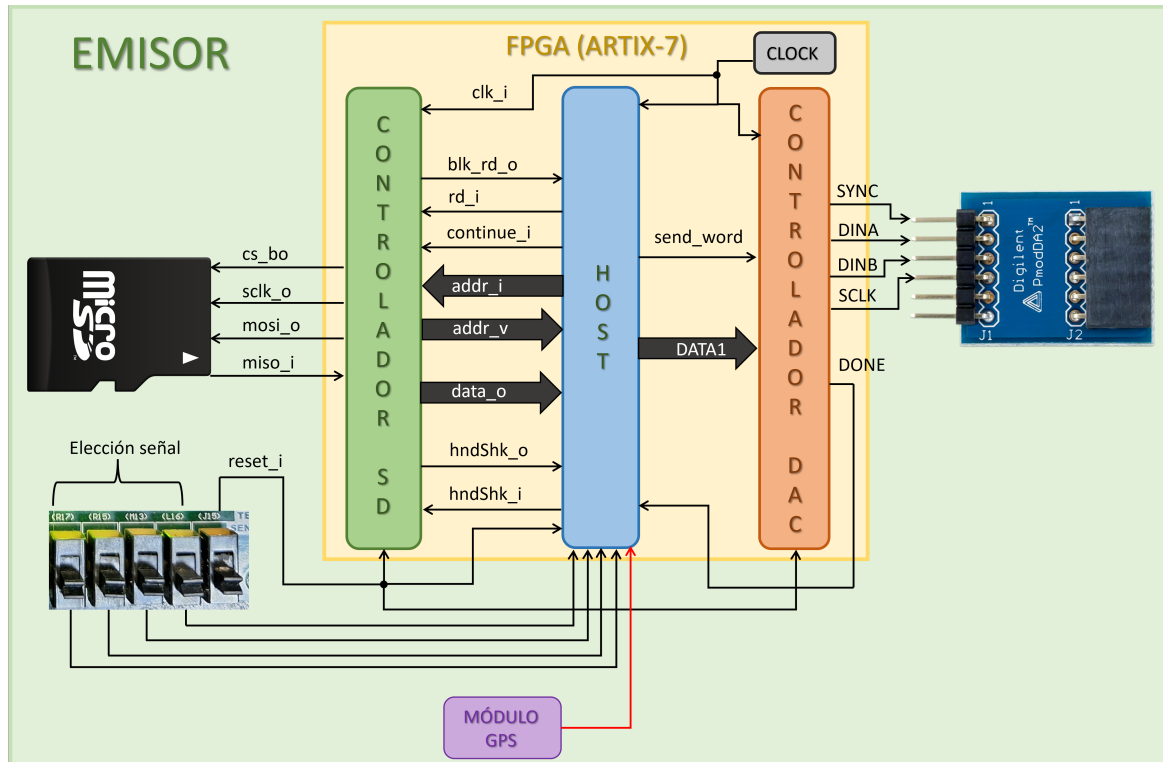


Figura 5.11: Diagrama en bloques simplificado del emisor.
Fuente: Elaboración propia.

Implementación

Para validar el correcto funcionamiento de este último bloque diseñado, se incorporó al componente, y solo a modo de pruebas, un generador de pulsos interno, que emule la señal con período de 1 segundo del módulo GPS. La implementación final de este bloque, no cuenta con el generador de pulsos mencionado, sino que se conectará a alguna de las entradas de los puertos Pmod disponibles de la Nexys 4, tal como se mencionó anteriormente. En la Figura 5.12 se presenta la repetición de la señal emitida en intervalos periódicos de 1 segundo.

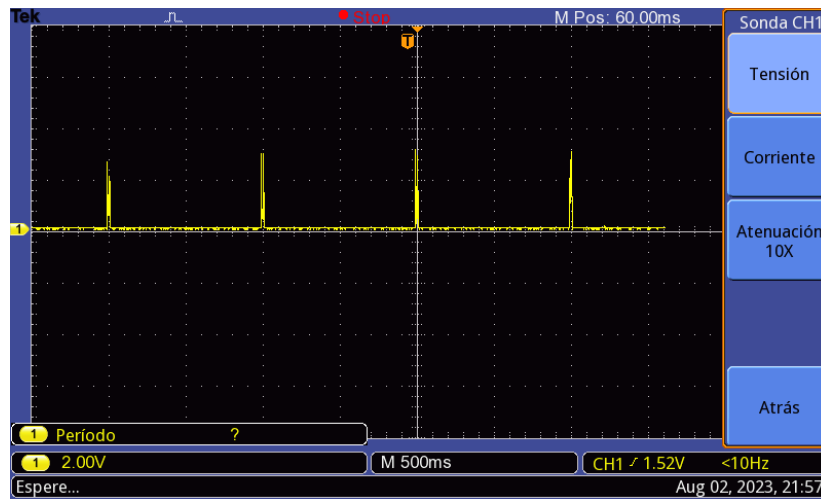


Figura 5.12: Captura del osciloscopio con la repetición de las señales cada 1 segundo.
Fuente: Elaboración Propia.

5.4. Resultados

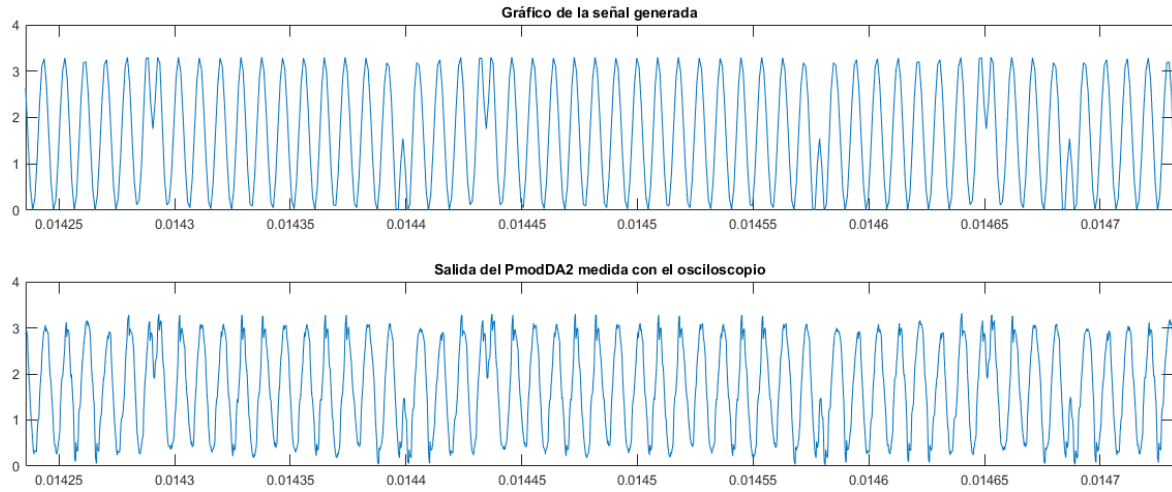
Como última prueba experimental, se almacenaron en la memoria SD 4 señales codificadas y moduladas de las presentadas en la sección 2.2 en el orden que se presenta a continuación:

1. BPSK-Kasami de longitud 1021.
2. PSK-Zadoff-Chu de longitud 1023.
3. OCDM-Zadoff-Chu de longitud 89.
4. OCDM-Zadoff-Chu de longitud 1423.

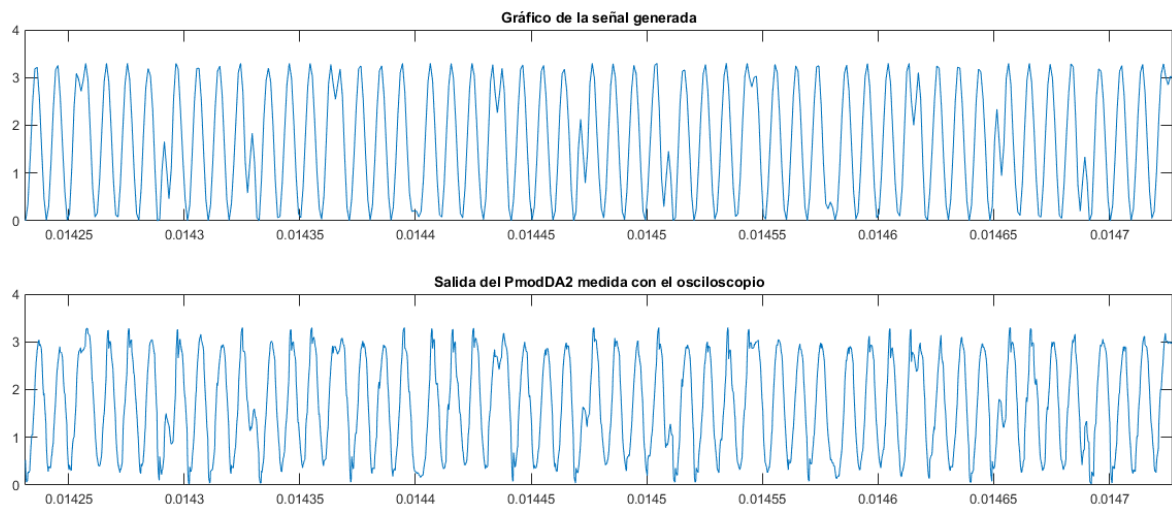
Posteriormente, se conectó el osciloscopio a la salida del PmodDA2 y se capturaron fragmentos de cada una de las secuencias, escogiéndolas por medio de los interruptores. Las capturas obtenidas del osciloscopio se almacenaron en un pen-drive para ser representadas en Matlab. Una vez procesadas las muestras, se las representó junto con los fragmentos de las señales reales para así poder compararlas.

A pesar de que se almacenaron las secuencias completas de las señales codificadas y moduladas, solo se capturaron fragmentos de las secuencias, puesto que al intentar obtener una captura completa, el osciloscopio modificaba su frecuencia de muestreo. De esta manera, al representar el conjunto de muestras almacenadas en el pen-drive, la señal resultante se encontraba muy distorsionada y resultaba complejo compararla con el conjunto de muestras de la señal original. Al capturar una menor cantidad de símbolos de cada señal, se pudo cotejar su semejanza con la señal real.

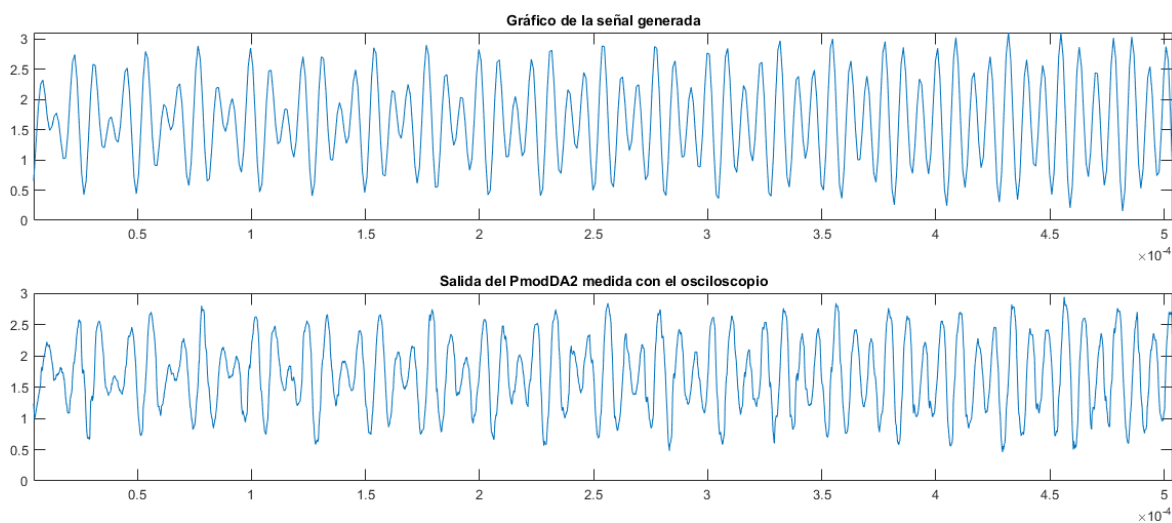
En las Figuras 5.13(d), 5.13(c) y 5.13(b) se presentan las gráficas de fragmentos de señales reales y señales a la salida del DAC medidas con el osciloscopio, para cada uno de los esquemas de codificación.



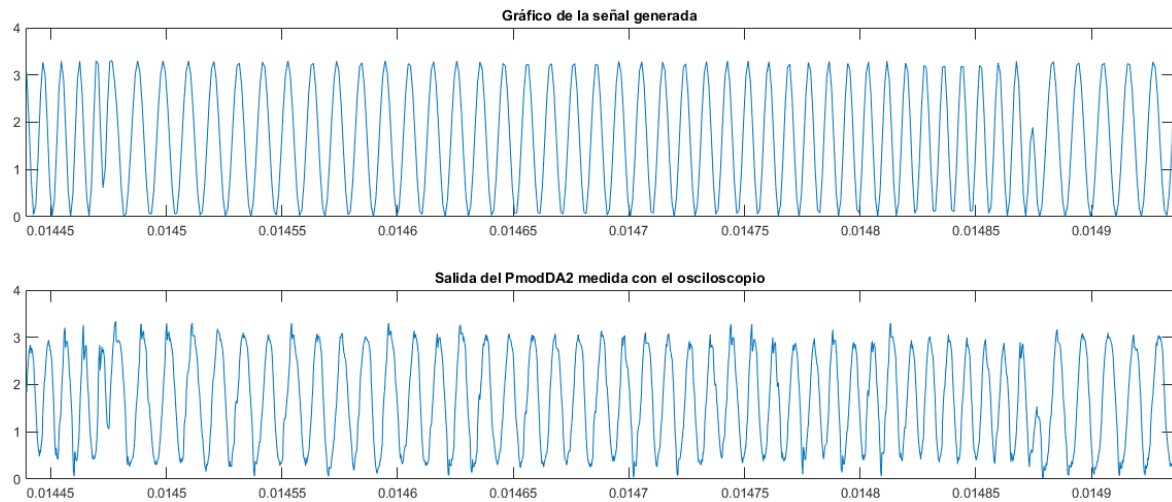
(a) Fragmento de Secuencia Kasami de longitud 1023 y modulación BPSK.



(b) Fragmento de Secuencia Zadoff-Chu de longitud 1021 y modulación PSK.



(c) Fragmento de Secuencia Zadoff-Chu de longitud 1423 y modulación OFDM.



(d) Fragmento de Secuencia Zadoff-Chu de longitud 89 y modulación OCDM.

Figura 5.13: Fragmentos las señales codificadas y moduladas.
Fuente: Elaboración Propia.

6. Conclusión

6.1. Conclusiones del proyecto

En el marco del Proyecto Final de carrera de Ingeniería Electrónica, se realizó el diseño e implementación de un dispositivo emisor de señales codificadas para sistemas de posicionamiento ultrasónico en entornos marinos. Este diseño implicó un riguroso estudio de sistemas de posicionamiento subacuáticos, desde sus aplicaciones, hasta los esquemas de codificación con buenas propiedades para la comunicación. De igual manera, se evaluaron las características del hardware a ser utilizado y se logró una familiarización con el software de diseño digital Vivado, al igual que con el lenguaje VHDL.

Para el almacenamiento de las señales codificadas y moduladas en la memoria SD, se utilizaron distintos software para la manipulación y formateo de las muestras. No obstante, cabe mencionar que los programas utilizados, tanto el HxD como el Notepad++, son de libre uso, lo cual permite que el usuario siga el procedimiento descrito sin inconvenientes. Por otro lado, el código en Matlab para la representación hexadecimal de las muestras, puede ser fácilmente implementado en Octave¹, que puede ser considerado el equivalente libre de Matlab.

Para lograr los objetivos planteados en alusión al desarrollo del emisor, se dividió el proyecto en etapas y se realizó el diseño de cada una de estas por separado. De esta manera, cada etapa pudo ser diseñada, simulada y verificada individualmente sin ser afectada por las demás. Esto permitió simplificar no solo el objetivo final de diseño, sino también la depuración de cada etapa antes de avanzar con la siguiente.

La verificación de las etapas del proyecto (sobre todo el controlador de memoria SD) resultó ser una tarea ardua y laboriosa. Esto no hubiera sido así si los códigos se hubieran realizado sin errores desde un principio. Al no cumplirse esto, los bloques diseñados no se comportaban según lo esperado y comenzaba un proceso de depuración que implicó recurrir a los recursos visuales (leds) de la Nexys 4 DDR para identificar el origen del problema. Los 16 leds presentes en la placa de desarrollo fueron fundamentales para verificar que las máquinas de estado del proyecto funcionaran según lo previsto, sin saltarse estados.

Una vez diseñados los dos grandes bloques del emisor (el controlador de memoria SD y el controlador del PmodDA2), se pudo diseñar la interfaz que los vinculara sin mayores inconvenientes. Si bien en un principio se planteó la posibilidad de realizar una interfaz directa, esta resultó ser no viable, motivo por el cual se escogió una interfaz con memoria intermedia. Esta memoria RAM intermedia cuenta con la limitación de poseer un almacenamiento fijado en el código. Por lo tanto, el usuario tendría que eventualmente modificarlo según sus

¹<https://octave.org>

requisitos. Sin embargo, tal como se mencionó previamente, se diseñó una memoria cuyo tamaño pudiera hacer frente a incrementos importantes en la longitud de las señales almacenadas. Adicionalmente, se fijó el tamaño de la memoria a través de constantes al inicio del código, por lo cual su modificación no es compleja.

Luego de realizar pruebas con señales con distintos esquemas de codificación y modulación se pudo verificar la similitud entre las señales originales y las representadas mediante el PmodDA2. En este sentido, es importante destacar que la frecuencia de muestreo del módulo PmodDA2 quedó fijada en 1 MSa/s, por lo cual la representación de señales con portadoras de 100 kHz (que es el caso de las señales del proyecto) es posible. La máxima frecuencia que puede ser transmitida con el emisor, según el Teorema de Nyquist-Shannon, es de 500 kHz. Por lo tanto, es aconsejable utilizar frecuencias de portadora inferiores a ese valor para evitar el aliasing.

6.2. Trabajo a futuro

A continuación, se mencionan algunas de las mejoras o incorporaciones adicionales que podrían sumarse al trabajo realizado.

Si bien el controlador de memoria SD se diseñó para que fuera compatible con memorias SDHC, se observó que no funcionó con memorias SDHC UHS-I. Sin embargo, los hosts compatibles con memorias SDHC tendrían que poder realizar operaciones con memorias SDHC UHS-I ². Por lo tanto, se tendría que estudiar con mayor detalle la incompatibilidad presentada e introducir los cambios pertinentes al controlador diseñado. También podría ampliarse el alcance del controlador para que soporte memorias SDXC y SDUC que cada vez se encuentran con mayor facilidad en el mercado.

Podría también añadirse el sistema de localización de archivos por medio de un sistema FAT (File-Allocations-Table), en lugar de acceder a los datos por medio de direcciones elementales de memoria. Esta opción toma relevancia si la aplicación particular cuenta con una mayor cantidad de señales a transmitir, lo cual complejiza el proceso de almacenamiento de las señales en la memoria.

Para emitir señales de mayor frecuencia se debe recurrir a otro módulo DAC que admita frecuencias de muestreo superiores a 1 MSa/s. El único módulo que cumple con este requisito, y que es compatible con la Nexys 4 DDR, es el PmodDA3, que soporta frecuencias de **SCLK** de hasta 50 MHz. No obstante, el protocolo de comunicación del módulo difiere del protocolo del PmodDA2. Por lo tanto, se tendría que rediseñar el controlador del conversor DAC.

Por último, podrían introducirse ajustes para transmitir a la salida del DAC, únicamente los valores de la RAM auxiliar que contienen muestras de la señal y no todo su contenido (junto con los espacios de memoria que almacenan ceros). Esto

²<https://www.integralmemory.com/faq1/what-is-uhs-1/>

podría ser necesario en caso de querer transmitir secuencias más cortas a menor periodicidad, es decir, en lugar de reenviar la secuencia cada 1 segundo, hacerlo por ejemplo cada 100 ms. El diseño actual admite el reenvío de la secuencia cada 200 ms (ver Figura [5.7\(b\)](#)).

Anexo

A continuación se presentan algunos de los códigos desarrollados en los capítulos anteriores. Para acceder a la totalidad del proyecto desarrollado en Vivado consultar con el Grupo de Investigación en Procesamiento de la Información y Sensores (GIPIS) del Departamento de Electrónica de la Universidad Nacional de la Patagonia San Juan Bosco (UNPSJB).

Controlador de memoria SD

Código 6.1: Controlador de memoria SD.

```
1  library IEEE, GEN;
2  use IEEE.std_logic_1164.all;
3  use IEEE.numeric_std.all;
4  use IEEE.math_real.all;
5  use IEEE.numeric_std.all;
6  use GEN.CommonPkg.all;
7
8  package SdcardPkg is
9  type CardType_t is (SD_CARD_E, SDHC_CARD_E); -- tipos de tarjetas SD
10
11 component SdCardCtrl is
12     generic(
13         FREQ_G          : real          := 100.0;    -- Frecuencia del
14         ↪ Master clock
15         INI_SPI_FREQ_G  : real          := 0.4;      -- Frecuencia de la
16         ↪ SPI durante la inicialización
17         SPI_FREQ_G      : real          := 25.0;     -- Frecuencia de
18         ↪ operación de la SPI
19         BLOCK_SIZE_G    : natural       := 512;      -- Número de bytes
20         ↪ en un bloque de la memoria SD
21         CARD_TYPE_G     : CardType_t    := SDHC_CARD_E -- Tipo de SD
22         ↪ conectada al controlador
23     );
24
25     port(
```

```

21      -- Señales del lado del Host
22      clk_i      : in  std_logic;
23      reset_i    : in  std_logic                      := '0';
↪    -- Reset síncrono activo ALTO
24      rd_i      : in  std_logic                      := '0';
↪    -- Solicitud de lectura activo ALTO
25      continue_i : in  std_logic                      := '0';
↪    -- Si V, aumenta el addr y continua R/W
26      addr_i    : in  std_logic_vector(31 downto 0) := x"00000000";
↪    -- Dirección del bloque
27      data_o    : out std_logic_vector(7  downto 0) := x"00";
↪    -- datos a leer desde un bloque
28      busy_o    : out std_logic;
↪    -- '1' cuando el controlador está ocupado ejecutando alguna operación
29      hndShk_i  : in  std_logic;
↪    -- '1' cuando el HOST tiene datos que enviar o ha recibido datos
30      hndShk_o  : out std_logic;                      --
↪    -- '1' cuando el CONTROLADOR tiene datos que enviar o ha recibido datos
31      --error_o : out std_logic_vector(15 downto 0) := (others =>
↪    '0');
32      state_o   : out std_logic_vector(1  downto 0) := (others => '0');
33      byte_o    : out std_logic_vector(1  downto 0) := (others => '0');
34      blk_rd_o  : out std_logic;
35      addr_v_o  : out std_logic_vector(31 downto 0);
36
37      -- Señales del lado de la SDcard
38      cs_bo     : out std_logic                      := '1';
↪    -- Chip select activo baja
39      sclk_o    : out std_logic                      := '0';
↪    -- Serial clock a la SD
40      mosi_o    : out std_logic                      := '1';
↪    -- Serial data output a la SD
41      miso_i    : in  std_logic                      := '0'
↪    -- Serial data input desde la SD
42  );
43  end component;
44  end package;
45
46

```

```

47 library IEEE, GEN, SDController;
48 use IEEE.std_logic_1164.all;
49 use IEEE.numeric_std.all;
50 use IEEE.math_real.all;
51 use GEN.CommonPkg.all;
52 use SDController.SDcardPkg.all;
53
54 entity SdCardCtrl is
55     generic(
56         FREQ_G          : real          := 100.0;    --
57         ↪ Frecuencia del Master clock
58         INI_SPI_FREQ_G  : real          := 0.4;      --
59         ↪ Frecuencia de la SPI durante la inicialización
60         SPI_FREQ_G      : real          := 25.0;     --
61         ↪ Frecuencia de operación de la SPI
62         BLOCK_SIZE_G    : natural       := 512;      --
63         ↪ Número de bytes en un bloque de la memoria SD
64         CARD_TYPE_G     : CardType_t    := SDHC_CARD_E
65     );
66
67     port(
68         -- Señales del lado del Host
69         clk_i      : in std_logic;
70         reset_i    : in std_logic      := '0';
71         rd_i       : in std_logic      := '0';
72         continue_i : in std_logic      := '0';
73         addr_i     : in std_logic_vector(31 downto 0) := x"00000000";
74         data_o     : out std_logic_vector(7 downto 0) := x"00";
75         busy_o     : out std_logic;
76         hndShk_i   : in std_logic;
77         hndShk_o   : out std_logic;
78         --error_o   : out std_logic_vector(15 downto 0) := (others =>
79         ↪ '0');
80         state_o    : out std_logic_vector(1 downto 0) := (others => '0');
81         byte_o     : out std_logic_vector(1 downto 0) := (others => '0');
82         blk_rd_o    : out std_logic;
83         addr_v_o    : out std_logic_vector(31 downto 0);
84
85         -- Señales del lado de la SD

```



```

81         cs_bo      : out std_logic      := '1';
82         sclk_o      : out std_logic      := '0';
83         mosi_o      : out std_logic      := '1';
84         miso_i      : in  std_logic      := '0'
85     );
86 end entity;
87
88 architecture arch of SdCardCtrl is
89     signal sclk_r    : std_logic := '0';
90     signal hndShk_r  : std_logic := '0';
91
92 begin
93
94     process(clk_i)
95
96         type FSM_t is (
97             START_INIT,           -- Se envían pulsos de inicialización a
98             ↪ la SD
99             SEND_CMD0,            -- Se pone a la SD en IDLE STATE
100             CHK_CMD0_RESP,        -- Se verifica la respuesta R1 de la SD
101             ↪ al comando CMD0
102             SEND_CMD8,            -- Se envía CMD8 a la SD (para la
103             ↪ inicialización de la SD)
104             GET_CMD8_RESP,        -- Se verifica la respuesta R7 de la SD
105             ↪ al comando CMD8
106             SEND_CMD55,           -- Se envía CMD55 a la SD
107             SEND_CMD41,           -- Se envía CMD41 a la SD
108             CHK_ACMD41_RESP,      -- Se verifica si la SD salió del IDLE
109             ↪ STATE
110             WAIT_FOR_HOST_RW,     -- Se espera que el host envíe un
111             ↪ comando de R o W
112             RD_BLK,               -- Lectura de un bloque de datos de la SD
113             START_TX,             -- Se comienza la transmisión de datos
114             ↪ o comandos
115             TX_BITS,
116             GET_CMD_RESP,         -- Se espera la respuesta R1 a un
117             ↪ comando por parte de la SD
118             RX_BITS,             -- Recepción de los datos de la SD

```

```

111         DESELECT,                                -- Se deselecciona la SD y se envían
↪ algunos pulsos de clock
112         PULSE_SCLK,
113         REPORT_ERROR                                -- Se reporta un error y se detiene
↪ hasta el reset
114     );
115     variable state_v      : FSM_t := START_INIT; -- Estado actual de
↪ la FSM
116     variable ret_state_v : FSM_t;                -- Estado al cual
↪ regresa la FSM cuando se completa la subrutina de la FSM
117
118     -- Constantes de tiempo basadas en la frecuencia del master clock
↪ y las frecuencias de clock de la SPI
119
120     constant CLKS_PER_INIT_SCLK_C      : real      := FREQ_G /
↪ INI_SPI_FREQ_G;
121     constant CLKS_PER_SCLK_C           : real      := FREQ_G /
↪ SPI_FREQ_G;
122     constant MAX_CLKS_PER_SCLK_C       : real      :=
↪ realmax(CLKS_PER_INIT_SCLK_C,CLKS_PER_SCLK_C);
123     constant MAX_CLKS_PER_SCLK_PHASE_C : natural :=
↪ integer(round(MAX_CLKS_PER_SCLK_C / 2.0));
124     constant INIT_SCLK_PHASE_PERIOD_C  : natural :=
↪ integer(round(CLKS_PER_INIT_SCLK_C) / 2.0);
125     constant SCLK_PHASE_PERIOD_C       : natural :=
↪ integer(round(CLKS_PER_SCLK_C) / 2.0);
126     constant DELAY_BETW_B'0'CK_RW_C   : natural :=
↪ SCLK_PHASE_PERIOD_C;
127
128     -- Registros para la generación del SPI SCLK (lento) a partir del
↪ master clock (veloz)
129     variable clk_Div_v                : natural range 0 to
↪ MAX_CLKS_PER_SCLK_PHASE_C;    -- Contiene el período de sclk
130     variable sclk_Phase_timer_v       : natural range 0 to
↪ MAX_CLKS_PER_SCLK_PHASE_C;    -- Cuenta hasta 0 y después se invierte
↪ sclk
131

```

```

132     constant NUM_INIT_CLKS_C      : natural := 160;
    ↪      -- Número de ciclos de clock para la inicialización de
    ↪  la SD
133     variable bit_Cnt_v           : natural range 0 to
    ↪  NUM_INIT_CLKS_C;             -- Contador de bits Tx/Rx
134     constant CRC_SZ_C           : natural := 2;
    ↪      -- Número de bytes de CRC para bloques R/W
135
136     -- Al leer bloques de datos: 0xFE + [bloque DE DATOS] + [CRC]
137     constant RD_BLK_SZ_C        : natural := 1 + BLOCK_SIZE_G +
    ↪  CRC_SZ_C;
138
139     -- Al escribir bloques de datos: 0xFF + 0xFE + [bloque DE DATOS] +
    ↪  [CRC] y luego recibir el byte de rta
140     constant WR_BLK_SZ_C        : natural := 1 + 1 + BLOCK_SIZE_G +
    ↪  CRC_SZ_C + 1;
141
142     variable byte_Cnt_v         : natural range 0 to
    ↪  IntMax(WR_BLK_SZ_C, RD_BLK_SZ_C);
143
144     -- Comandos
145     subtype Cmd_t is std_logic_vector(7 downto 0);
146     constant CMD0_C             : Cmd_t :=
    ↪  std_logic_vector(to_unsigned(16#40# + 0, Cmd_t'length));
147     constant CMD8_C             : Cmd_t :=
    ↪  std_logic_vector(to_unsigned(16#40# + 8, Cmd_t'length));
148     constant CMD55_C            : Cmd_t :=
    ↪  std_logic_vector(to_unsigned(16#40# + 55, Cmd_t'length));
149     constant CMD41_C            : Cmd_t :=
    ↪  std_logic_vector(to_unsigned(16#40# + 41, Cmd_t'length));
150     constant READ_BLK_CMD_C      : Cmd_t :=
    ↪  std_logic_vector(to_unsigned(16#40# + 17, Cmd_t'length));
151
152     -- Las operaciones que necesitan CRC son CMD0 y CMD8 por lo cual
    ↪  para las demás se utiliza un CRC falso
153     constant FAKE_CRC_C : std_logic_vector(7 downto 0) := x"FF";
154
155     variable addr_v : unsigned(addr_i'range); -- Dirección del bloque
    ↪  actual para R/W

```

```

156
157     -- Variables de transmisión
158     variable tx_v :
    ↪ std_logic_vector(CMD0_C'length+addr_v'length+FAKE_CRC_C'length-1
    ↪ downto 0);
159     alias tx_Cmd_v is tx_v;
160     alias tx_Data_v is tx_v(tx_v'high downto tx_v'high -
    ↪ data_o'length + 1);
161
162     -- Variables de recepción
163     variable rx_v : std_logic_vector(data_o'range);
164
165     -- Códigos de respuesta
166     subtype response_t is std_logic_vector(rx_v'range);
167     constant ACTIVE_'0'_ERRORS_C : response_t := "00000000"; -- Rta
    ↪ normal R1 después de la inicialización
168     constant IDLE_'0'_ERRORS_C : response_t := "00000001"; -- Rta
    ↪ normal R después del comando CMD0
169     constant DATA_ACCEPTED_C : response_t := "---00101"; -- La SD
    ↪ acepta el bloque de datos
170     constant DATA_REJ_CRC_C : response_t := "---01011"; -- La SD
    ↪ rechaza el bloque de datos del host como consecuencia de un error en
    ↪ el CRC
171     constant DATA_REJ_WERR_C : response_t := "---01101"; -- La SD
    ↪ rechaza el bloque de datos como consecuencia de un error en el CRC
172
173     -- Tokens
174     subtype token_t is std_logic_vector(rx_v'range);
175     constant NO_TOKEN_C : token_t := x"FF"; -- Es recibido antes
    ↪ de que la SD responda a un comando de lectura de bloque
176     constant START_TOKEN_C : token_t := x"FE"; -- Byte de inicio
    ↪ que precede a un bloque de datos
177
178     -- Flags que se setean o se "limpian" durante la operación de la FSM
179     variable getCmdResponse_v : boolean; -- Cuando es V, se obtiene
    ↪ la respuesta R1 de un comando enviado a la SD
180     variable ret_Data_v : boolean; --
181     variable dDeselect_v : boolean; -- Cuando es V, se
    ↪ deselecta la SD después de que se ejecute un comando

```

```

182
183
184 -- FSM
185
186 begin
187     if rising_edge(clk_i) then
188
189         if reset_i = '1' then -- Resetear
190             state_v := START_INIT; -- No se
191             busy_o <= '1'; -- Mientras la
192             byte_o <= "00";
193
194             elsif sclk_Phase_timer_v /= 0 then
195                 -- Generamos un sclk de período mayor que el master clock
196                 sclk_Phase_timer_v := sclk_Phase_timer_v - 1;
197
198                 -- En un momento dado el phase timer llega a cero y hay que
199                 -- verificar el handshaking entre el host y el controlador
200
201                 -- El handshake permite que el host controle el flujo de
202                 -- datos hacia/desde el controlador de SD
203                 -- El handshake se ejecuta de la siguiente manera:
204
205                 -- 1) El controlador levanta su hndShk y espera (hndShk_r
206                 -- = '1').
207
208                 -- 2) El host recibe el hndShk del controlador y levanta su
209                 -- hndShk a modo de ACK (hndShk_i = '1').
210
211                 -- 3) El controlador ve el ACK hndShk del host y baja su
212                 -- hndShk (hndShk_r = '0').
213
214                 -- 4) El host ve que el controlador bajó su hndShk y hace
215                 -- lo mismo (hndShk_i = '0').
216
217             elsif state_v /= START_INIT and hndShk_r = '1' and hndShk_i =
218             '0' then
219                 null; -- Se espera el host ACK del hndShk
220             elsif state_v /= START_INIT and hndShk_r = '1' and hndShk_i =
221             '1' then

```

```

211             hndShk_r <= '0';
212         elsif state_v /= START_INIT and hndShk_r = '0' and hndShk_i =
↪ '1' then
213             null; -- Se espera que el host baje su
↪ hndShk
214         elsif (state_v = START_INIT) or (hndShk_r = '0' and hndShk_i =
↪ '0') then
215
216             -- Los dos hndSk están en bajo por lo que el controlador
↪ comienza a operar
217
218             busy_o <= '1'; -- Comienzan las operaciones por lo que el
↪ controlador estará en busy
219
220             case state_v is
221
222                 when START_INIT => -- Des-seleccionar la SD y enviar
↪ pulsos de clock con MOSI = '1'
223                     --error_o <= (others => '0');
↪ -- Limpiamos las flags de error
224                     clk_Div_v := INIT_SCLK_PHASE_PERIOD_C
↪ - 1; -- Durante la incialización usamos una frecuencia del clock
↪ SPI lenta
225                     sclk_Phase_timer_v := INIT_SCLK_PHASE_PERIOD_C
↪ - 1; -- Se setea la duración de la poróxima fase del clock
226                     sclk_r <= '0';
↪ -- Iniciamos el clock en bajo
227                     hndShk_r <= '0';
↪ -- Iniciamos la señal hndShk
228                     addr_v := (others => '0');
↪ -- Iniciamos la dirección
229                     ret_Data_v := false;
↪ -- no se devuelve data al host durante la inicialización
230                     bit_Cnt_v := NUM_INIT_CLKS_C;
↪
231                     state_v := DESELECT;
232                     ret_state_v := SEND_CMD0;
↪ -- ir a este estado una vez que finalizaron los pulsos de clock
233

```

```

234         when SEND_CMD0 => -- Poner la SD en el IDLE STATE
235             cs_bo                <= '0';
↪         -- Se habilita la SD
236             tx_Cmd_v              := CMD0_C & x"00000000" &
↪         x"95" ; -- 0x95 es el CRC correcto para este comando
237             bit_Cnt_v             := tx_Cmd_v'length;
↪         -- Se setea el contador de bits al tamaño del comando
238             getCmdResponse_v       := true;
↪         -- El comando genera un resapuesta
239             dDeselect_v            := true;
↪         -- Se des-selecciona la SD
240             state_v                := START_TX;
↪         -- Ir a esta subrutina de la FSM para enviar el comando
241             ret_state_v             := CHK_CMD0_RESP;
↪         -- Verifica la respuesta de la SD al comando
242
243         when CHK_CMD0_RESP => -- Se verifica la respuesta R1
↪         al comando CMD0
244             if rx_v = IDLE_'0'_ERRORS_C then
245                 state_v              := SEND_CMD8;
↪         -- Si no hay errores continúa la inicialización
↪
246             else
247                 state_v              := SEND_CMD0;
↪         -- Caso contrario se vuelve a enviar CMD0
248             end if;
249
250         when SEND_CMD8 => -- Comando requerido para
↪         inicializar SDHCs
251             cs_bo                <= '0';
↪         -- Se habilita la SD
252             tx_Cmd_v              := CMD8_C & x"000001aa" &
↪         x"87"; -- 0x87 es el CRC correcto para este comando
253             bit_Cnt_v             := tx_Cmd_v'length;
↪         -- Se setea el contador de bits al tamaño del comando
254             getCmdResponse_v       := true;
↪         -- Este comando genera una respuesta
255             dDeselect_v            := false;
↪         -- no se des-selecciona la SD para esperar la respuesta R7

```

```

256             state_v                := START_TX;
257             ret_state_v             := GET_CMD8_RESP;
↪             -- Luego de enviar el comando vamos a este estado
258
259             when GET_CMD8_RESP => -- Se verifica la respuesta R7
↪             al comando CMD8
260                 cs_bo                <= '0';
↪                 -- La Sd ya está habilitada de todas maneras
261                 bit_Cnt_v            := 31;
↪                 -- Cuatro bytes (32 bits) en la respuesta R7
262                 getCmdResponse_v     := false;
↪                 -- El comando '0' genera una respuesta
263                 dDeselect_v          := true;
↪                 -- Des-seleccionar la tarjeta SD para finalizar el comando
↪                 después de recibir los 4 bytes
264                 state_v              := RX_BITS;
↪                 -- Va a la subrutina de FSM para recibir la respuesta R7
265                 ret_state_v          := SEND_CMD55;
↪                 -- Va a la subrutina indicada (no importa la respuesta R7,
↪                 importa que sea recibida)
266
267             when SEND_CMD55 =>      -- Se envía CMD55 como
↪             preámbulo de ACMD41
268                 cs_bo                <= '0';
269                 tx_Cmd_v             := CMD55_C & x"00000000" &
↪             FAKE_CRC_C;
270                 bit_Cnt_v            := tx_Cmd_v'length;
271                 getCmdResponse_v     := true;
↪                 -- Se envía un comando que genera respuesta
272                 dDeselect_v          := true;
↪                 -- Des-seleccionar
273                 state_v              := START_TX;
274                 ret_state_v          := SEND_CMD41;
275
276             when SEND_CMD41 =>
277                 cs_bo                <= '0';
278                 tx_Cmd_v             := CMD41_C & x"40000000" &
↪             FAKE_CRC_C;
279                 bit_Cnt_v            := tx_Cmd_v'length;

```



```

280         getCmdResponse_v           := true;
281         dDeselect_v                 := true;
282         state_v                     := START_TX;
283         ret_state_v                 := CHK_ACMD41_RESP;
284
285         when CHK_ACMD41_RESP =>
286             -- Los comandos CMD55 y CMD41 deberían hacer que la SD
287             -- salga del IDLE STATE
288             -- Adicionalmente la SD debe quedar lista para para
289             -- realizar operaciones SPI R/W
290             -- Si la SD continúa en estado IDLE hay que repetir las
291             -- secuencias CMD55 y CMD41
292             if rx_v = ACTIVE_'0'_ERRORS_C then
293                 state_v := WAIT_FOR_HOST_RW;
294             elsif rx_v = IDLE_'0'_ERRORS_C then
295                 state_v := SEND_CMD55;
296             else
297                 state_v := REPORT_ERROR;
298             end if;
299
300         when WAIT_FOR_HOST_RW => -- Esperar a que el host
301             -- escriba o lea un bloque de datos de la SD
302             state_o <= "01";
303             clk_Div_v := SCLK_PHASE_PERIOD_C - 1 ;
304             getCmdResponse_v := true;
305             -- Obtener la respuesta R1
306             blk_rd_o <= '0';
307             if rd_i = '1' then -- Hay que enviar el comando
308             -- READ y la dirección a la SD
309                 cs_bo <= '0';
310             -- Habilidad de la SD
311                 if continue_i = '1' then
312             -- Lectura de muchos bloques
313                     if CARD_TYPE_G = SD_CARD_E then
314             -- Direcccionamiento por bytes
315                         addr_v := addr_v + BLOCK_SIZE_G;
316             -- Para obtener la próxima dirección le sumo el tamaño del bloque
317                     else
318             -- Direcccionamiento por bloques entonces incremento el bloque actual

```

```

308             addr_v             := addr_v + 1;
↪
309             end if ;
310             tx_Cmd_v             := READ_BLK_CMD_C &
↪ std_logic_vector(addr_v) & FAKE_CRC_C;
311             else
↪             -- Lectura de un único bloque
312             tx_Cmd_v             := READ_BLK_CMD_C & addr_i &
↪ FAKE_CRC_C;    -- Se utiliza la dirección provista por el host
313             addr_v             := unsigned(addr_i);
↪             -- Guardo la dirección para operaciones de múltiples bloques
314             end if ;
315             bit_Cnt_v            := tx_Cmd_v'length;
316             byte_Cnt_v          := RD_BLK_SZ_C;
317             state_v             := START_TX;
318             ret_state_v         := RD_BLK;
319             addr_v_o            <= std_logic_vector(addr_v);
320             else
↪             -- No hacer nada y esperar algún comando del host
321             cs_bo                <= '1';
↪             -- Des-seleccionar la SD
322             busy_o              <= '0';
↪             -- La interfaz se queda esperando R/W del host (no está ocupada)
323             state_v            := WAIT_FOR_HOST_RW;
↪             -- Sigue esperando un comando del host
324             end if;
325
326             when RD_BLK =>
327             -- Valores por defecto
328             ret_Data_v          := false;
↪             --
329             bit_Cnt_v          := rx_v'length - 1;
330             state_v            := RX_BITS;
331             ret_state_v        := RD_BLK;
332
333             if byte_Cnt_v = RD_BLK_SZ_C then
334             byte_Cnt_v          := byte_Cnt_v - 1;
335             elsif byte_Cnt_v = RD_BLK_SZ_C - 1 then
↪             -- Buscar el token de inicio de datos

```

```

336         if rx_v = '0'_TOKEN_C then
→      -- no se encontró el token todavía
337             null;
338         elsif rx_v = START_TOKEN_C then
339             ret_Data_v          := true;
→      -- Se encontró el token de inicio de datos => se comienzan a
→      enviar datos al host
340             byte_Cnt_v          := byte_Cnt_v - 1;
341         else
→      -- Cualquier otra situación implica que ocurrió algo extraño
342             state_v              := REPORT_ERROR;
343         end if ;
344         elsif byte_Cnt_v >=3 then
345             ret_Data_v            := true;
346             byte_Cnt_v            := byte_Cnt_v - 1;
347         elsif byte_Cnt_v = 2 then
→      -- 1er byte de CRC
348             byte_Cnt_v            := byte_Cnt_v - 1;
349             byte_o                <= "10";
350         elsif byte_Cnt_v = 1 then
→      -- 2ndo byte de CRC
351             byte_Cnt_v            := byte_Cnt_v - 1;
352             byte_o                <= "01";
353         else
354             sclk_r                <= '0';
355             byte_o                <= "11";
356             blk_rd_o              <= '1';
357             bit_Cnt_v             := 2;
358             state_v               := DESELECT;
359             ret_state_v           := WAIT_FOR_HOST_RW;
360         end if;
361
362         when START_TX => -- Se comienza enviando comandos/datos
→      bajando SCLK y poniendo en la salida el MSB del comando/dato
363             sclk_r                <= '0';
364             sclk_Phase_timer_v     := clk_Div_v;          --
→      setear la duración del '0'W SCLK
365             mosi_o                <= tx_v(tx_v'high);    --
→      Colocar en la salida el MSB de comando o dato

```

```

366         tx_v                                     := tx_v(tx_v'high - 1 downto
→ 0) & '1' ; -- Desplazamos el contenido del registro 1 bit
367         bit_Cnt_v                               := bit_Cnt_v - 1;          --
→ El primer bit ya ha sido enviado
368         state_v                                 := TX_BITS;
369
370         when TX_BITS => -- Se van desplazando los datos o
→ comandos restantes y posiblemente se obtiene una respuesta de la SD
371         sclk_r                                   <= not sclk_r;
372         sclk_Phase_timer_v                       := clk_Div_v;
373         if sclk_r = '1' then
374             if bit_Cnt_v /= 0 then -- Seguir mandando bits
→ hasta que sea 0
375                 mosi_o                           <= tx_v(tx_v'high);
376                 tx_v                               := tx_v(tx_v'high - 1 downto
→ 0) & '1';
377                 bit_Cnt_v                         := bit_Cnt_v - 1;
378             else
379                 if getCmdResponse_v then
380                     state_v                       := GET_CMD_RESP;
381                     bit_Cnt_v                     := response_t'length -
→ 1; -- longitud de la respuesta esperada
382                 else
383                     state_v                       := ret_state_v;
384                     sclk_Phase_timer_v           := 0;
385                 end if ;
386             end if ;
387         end if ;
388
389         when GET_CMD_RESP =>
390             if sclk_r = '1' and miso_i = '0' then -- miso se
→ mantiene en HGI hasta el 1er bit de la rta R1 que es 0
391                 --desplazar el 1er bit de la rta
392                 rx_v                               := rx_v(rx_v'high -1
→ downto 0) & miso_i;
393                 bit_Cnt_v                         := bit_Cnt_v - 1;
394                 state_v                           := RX_BITS;          --
→ Recibo el resto de la rta
395             end if ;

```

```

396         sclk_r                                     <= not sclk_r;
397         sclk_Phase_timer_v                           := clk_Div_v;
398
399         when RX_BITS =>
400             if sclk_r = '1' then
401                 rx_v                                     := rx_v(rx_v'high - 1
↳         downto 0) & miso_i;
402                 if bit_Cnt_v /= 0 then
403                     bit_Cnt_v                           := bit_Cnt_v - 1;
404                 else                                     --
↳         El último bit ya fue recibido
405                     if ret_Data_v then
↳         -- Enviar los datos recibidos al host
406                         data_o                             <= rx_v;
↳         -- Poner los datos recibidos en la salida al host
407                         hndShk_r                           <= '1';
↳         -- Se avisa al host que los datos están listos
408                     end if ;
409                     if dDeselect_v then
410                         bit_Cnt_v                           := 1;
411                         state_v                             := DESELECT;
412                     else
413                         state_v                             := ret_state_v;
414                     end if ;
415                 end if ;
416             end if ;
417             sclk_r                                     <= not sclk_r;
418             sclk_Phase_timer_v                           := clk_Div_v;
419
420             when DESELECT =>                             -- Des-seleccionar la SD y
↳         enviar algunos pulsos de clock
421                 dDeselect_v                             := false;
↳         -- Borramos el flag que hizo que se des-seleccionara
422                 cs_bo                                     <= '1';
423                 mosi_o                                     <= '1';
424                 state_v                                   := PULSE_SCLK;
425                 sclk_r                                     <= '0';
426                 sclk_Phase_timer_v                       := clk_Div_v;
427

```

```

428         when PULSE_SCLK =>
429             if sclk_r = '1' then
430                 if bit_Cnt_v /= 0 then
431                     bit_Cnt_v := bit_Cnt_v - 1;
432                 else
433                     state_v := ret_state_v;
434                 end if ;
435             end if ;
436             sclk_r      <= not sclk_r;
437             sclk_Phase_timer_v := clk_Div_v;
438
439             when REPORT_ERROR =>
440                 state_o      <= "10";
441                 --error_o(rx_v'range) <= rx_v;
442                 busy_o      <= '0';
443
444             when others =>
445                 state_v      := START_INIT;
446
447         end case;
448     end if;
449 end if;
450 end process;
451
452     sclk_o      <= sclk_r;
453     hndShk_o    <= hndShk_r;
454
455 end architecture;
456
457
458
459
460

```

Controlador del PmodDA2

Código 6.2: Controlador del PmodDA2.

```
1  library IEEE, PMODDAC;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.numeric_std.all;
4  use PMODDAC.PmodDa2Pkg.all;
5
6  entity PModDa2 is
7      Port (
8          -- Entradas del controlador
9          CLK      : in std_logic;
10         reset_i  : in std_logic;
11         DATA1   : in std_logic_vector(15 downto 0);
12         DATA2   : in std_logic_vector(15 downto 0);
13         START    : in std_logic;
14         SCLK     : in std_logic;
15         -- Salidas del controlador
16         SYNC     : out std_logic;
17         DONE     : out std_logic;
18         DINA     : out std_logic;
19         DINB     : out std_logic
20     );
21 end entity;
22
23 architecture DA2 of PModDa2 is
24
25
26     -- SEÑALES
27     type states is (Idle_state, ShiftOut_state, SyncData_state); -- Distintos
28     -- estados para la máquina de estados
29     signal state_s      : states;          -- Estado actual
30     signal ret_state_s  : states;          -- Estado siguiente
31
32     signal temp1_s      : std_logic_vector(15 downto 0);
33     signal temp2_s      : std_logic_vector(15 downto 0);
34     signal clk_counter  : integer;
35     signal shiftCounter_s : unsigned(4 downto 0);
```

```

35  signal enShiftCounter_s : std_logic;           -- (enable) Shift Counter
36  signal enParalelLoad_s  : std_logic;           -- (enable) Paralel Load
37
38  begin
39
40  -----
41  ----- CONTADOR & DESPLAZAMIENTO
42  -----
43  -- Se cargan y desplazan los registros temporales, se lleva una cuenta de
44  -- los bits desplazados ----
45  -----
46  counter_process :
47  -- process(CLK,SCLK,enParalelLoad_s,enShiftCounter_s,reset_i)
48  begin
49  if reset_i = '1' then
50  shiftCounter_s <= "00000";
51  temp1_s        <= x"0000";
52  temp2_s        <= x"0000";
53  elsif rising_edge(SCLK) then
54      if (enParalelLoad_s = '1') then
55          shiftCounter_s <= "00000";
56          temp1_s <= DATA1;
57          temp2_s <= DATA2;
58          elsif (enShiftCounter_s = '1') then
59              temp1_s <= temp1_s(14 downto 0)&temp1_s(15);
60              temp2_s <= temp2_s(14 downto 0)&temp2_s(15);
61              shiftCounter_s <= shiftCounter_s + 1;
62          end if;
63  end if;
64  end process;
65
66  DINA <= temp1_s(15);
67  DINB <= temp2_s(15);
68
69  -----
70  ----- SYNC
71  -----
72  --
73  -----
74  SYNC_PROC: process (clk,SCLK,reset_i)

```



```

70     begin
71     if reset_i = '1' then
72     state_s <= Idle_state;
73     elsif rising_edge(SCLK) then
74         state_s <= ret_state_s;
75     end if;
76     end process;
77
78 -----
79 ----- MÁQUINA DE ESTADOS (MOORE)
80 ----- Se actualiza el valor de la salida a partir del estado
81 -----
82
83 FSM_state : process(state_s, reset_i)
84
85 begin
86 if reset_i = '1' then
87 enShiftCounter_s <= '0';
88 SYNC              <= '1';
89 DONE              <= '1';
90 enParallelLoad_s  <= '1';
91
92 else
93     case state_s is
94         when Idle_state =>
95             enShiftCounter_s <= '0';
96             DONE              <= '1';
97             SYNC              <= '1';
98             enParallelLoad_s  <= '1';
99
100         when ShiftOut_state =>
101             enShiftCounter_s <= '1';
102             DONE              <= '0';
103             SYNC              <= '0';
104             enParallelLoad_s  <= '0';
105
106         when SyncData_state =>

```

```

107         enShiftCounter_s <= '0';
108         DONE             <= '0';
109         SYNC             <= '1';
110         enParallelLoad_s <= '0';
111
112         when others => null;
113     end case;
114 end if;
115 end process;
116
117 -----
118 ----- GENERACION DEL ESTADO SIGUIENTE
119     ⇨ -----
120 ----- Se actualiza el valor del siguiente estado a partir del estado
121     ⇨ actual y las entradas -----
122 -----
123
122 ret_state_process : process(state_s,START,shiftCounter_s)
123 --ret_state_process : process(state_s,START,shiftCounter_s)
124 begin
125     ret_state_s <= state_s;  -- me quedo en el estado actual por defecto
126     case state_s is
127         when Idle_state =>
128             if START = '1' then
129                 ret_state_s <= ShiftOut_state;
130             else
131                 null;
132             end if;
133
134         when ShiftOut_state =>
135             if shiftCounter_s = "10000" then
136                 ret_state_s <= SyncData_state;
137             else
138                 null;
139             end if;
140
141         when SyncData_state =>
142             if START = '0' then
143                 ret_state_s <= Idle_state;

```

```

144         else
145         null;
146         end if;
147
148         when others =>
149             ret_state_s <= Idle_state;
150
151         end case;
152     end process;
153
154
155 end architecture;
156

```

MMCME2_ADV

Código 6.3: Instanciación del MMCME2_ADV para la obtención del reloj de 20 MHz a partir del de 100 MHz.

```

1  library IEEE, UNISIM;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_ARITH.ALL;
4  use IEEE.STD_LOGIC_UNSIGNED.ALL;
5  use UNISIM.vcomponents.all;
6
7  entity ClkDiv is
8      Port ( clk100 : in STD_LOGIC;
9            clk20  : out STD_LOGIC);
10 end ClkDiv;
11
12 architecture Behavioral of ClkDiv is
13     signal pll_locked : std_logic;
14     signal clkfbout : std_logic;
15     signal clk20_s : std_logic;
16 begin
17     MMCME2_ADV_inst : MMCME2_ADV
18     generic map (

```

```

19     BANDWIDTH => "OPTIMIZED",      -- Jitter programming (OPTIMIZED,
↪   HIGH, LOW)
20     CLKFBOUT_MULT_F => 12.0,      -- Multiply value for all CLKOUT
↪   (2.000-64.000).
21     CLKFBOUT_PHASE => 0.0,        -- Phase offset in degrees of CLKFB
↪   (-360.000-360.000).
22     -- CLKIN_PERIOD: Input clock period in ns to ps resolution (i.e.
↪   33.333 is 30 MHz).
23     CLKIN1_PERIOD => 10.000,
24     CLKIN2_PERIOD => 0.0,
25     -- CLKOUT0_DIVIDE - CLKOUT6_DIVIDE: Divide amount for CLKOUT (1-128)
26     CLKOUT1_DIVIDE => 30,
27     CLKOUT2_DIVIDE => 1,
28     CLKOUT3_DIVIDE => 1,
29     CLKOUT4_DIVIDE => 1,
30     CLKOUT5_DIVIDE => 1,
31     CLKOUT6_DIVIDE => 1,
32     CLKOUT0_DIVIDE_F => 1.0,      -- Divide amount for CLKOUT0
↪   (1.000-128.000).
33     -- CLKOUT0_DUTY_CYCLE - CLKOUT6_DUTY_CYCLE: Duty cycle for CLKOUT
↪   outputs (0.01-0.99).
34     CLKOUT0_DUTY_CYCLE => 0.5,
35     CLKOUT1_DUTY_CYCLE => 0.5,
36     CLKOUT2_DUTY_CYCLE => 0.5,
37     CLKOUT3_DUTY_CYCLE => 0.5,
38     CLKOUT4_DUTY_CYCLE => 0.5,
39     CLKOUT5_DUTY_CYCLE => 0.5,
40     CLKOUT6_DUTY_CYCLE => 0.5,
41     -- CLKOUT0_PHASE - CLKOUT6_PHASE: Phase offset for CLKOUT outputs
↪   (-360.000-360.000).
42     CLKOUT0_PHASE => 0.0,
43     CLKOUT1_PHASE => 0.0,
44     CLKOUT2_PHASE => 0.0,
45     CLKOUT3_PHASE => 0.0,
46     CLKOUT4_PHASE => 0.0,
47     CLKOUT5_PHASE => 0.0,
48     CLKOUT6_PHASE => 0.0,
49     CLKOUT4_CASCADE => FALSE,     -- Cascade CLKOUT4 counter with
↪   CLKOUT6 (FALSE, TRUE)

```

```

50     COMPENSATION => "ZHOLD",           -- ZHOLD, BUF_IN, EXTERNAL, INTERNAL
51     DIVCLK_DIVIDE => 2,                -- Master division value (1-106)
52     -- REF_JITTER: Reference input jitter in UI (0.000-0.999).
53     REF_JITTER1 => 0.0,
54     REF_JITTER2 => 0.0,
55     STARTUP_WAIT => FALSE,            -- Delays DONE until MMCM is locked
    => (FALSE, TRUE)
56     -- Spread Spectrum: Spread Spectrum Attributes
57     SS_EN => "FALSE",                 -- Enables spread spectrum (FALSE,
    => TRUE)
58     SS_MODE => "CENTER_HIGH",         -- CENTER_HIGH, CENTER_LOW,
    => DOWN_HIGH, DOWN_LOW
59     SS_MOD_PERIOD => 10000,           -- Spread spectrum modulation period
    => (ns) (VALUES)
60     -- USE_FINE_PS: Fine phase shift enable (TRUE/FALSE)
61     CLKFBOUT_USE_FINE_PS => FALSE,
62     CLKOUT0_USE_FINE_PS => FALSE,
63     CLKOUT1_USE_FINE_PS => FALSE,
64     CLKOUT2_USE_FINE_PS => FALSE,
65     CLKOUT3_USE_FINE_PS => FALSE,
66     CLKOUT4_USE_FINE_PS => FALSE,
67     CLKOUT5_USE_FINE_PS => FALSE,
68     CLKOUT6_USE_FINE_PS => FALSE
69 )
70 port map (
71     -- Clock Outputs: 1-bit (each) output: User configurable clock
    => outputs
72     CLKOUT0 => open,                  -- 1-bit output: CLKOUT0
73     CLKOUT0B => open,                 -- 1-bit output: Inverted CLKOUT0
74     CLKOUT1 => clk20_s,               -- 1-bit output: CLKOUT1
75     CLKOUT1B => open,                 -- 1-bit output: Inverted CLKOUT1
76     CLKOUT2 => open,                  -- 1-bit output: CLKOUT2
77     CLKOUT2B => open,                 -- 1-bit output: Inverted CLKOUT2
78     CLKOUT3 => open,                  -- 1-bit output: CLKOUT3
79     CLKOUT3B => open,                 -- 1-bit output: Inverted CLKOUT3
80     CLKOUT4 => open,                  -- 1-bit output: CLKOUT4
81     CLKOUT5 => open,                  -- 1-bit output: CLKOUT5
82     CLKOUT6 => open,                  -- 1-bit output: CLKOUT6
83     DO => open,                       -- 16-bit output: DRP data

```

```

84     DRDY => open,                -- 1-bit output: DRP ready
85     -- Dynamic Phase Shift Ports: 1-bit (each) output: Ports used for
    => dynamic phase shifting of the outputs
86     PSDONE => open,              -- 1-bit output: Phase shift done
87     -- Feedback Clocks: 1-bit (each) output: Clock feedback ports
88     CLKFBOUT => CLKFBOUT,        -- 1-bit output: Feedback clock
89     CLKFBOUTB => open,           -- 1-bit output: Inverted CLKFBOUT
90     -- Status Ports: 1-bit (each) output: MMCM status ports
91     CLKFBSTOPPED => open, -- 1-bit output: Feedback clock stopped
92     CLKINSTOPPED => open, -- 1-bit output: Input clock stopped
93     LOCKED => open,              -- 1-bit output: LOCK
94     -- Clock Inputs: 1-bit (each) input: Clock inputs
95     CLKIN1 => clk100,            -- 1-bit input: Primary clock
96     CLKIN2 => '0',              -- 1-bit input: Secondary clock
97     -- Control Ports: 1-bit (each) input: MMCM control ports
98     CLKINSEL => '1',            -- 1-bit input: Clock select, High=CLKIN1
    => Low=CLKIN2
99     PWRDWN => '0',              -- 1-bit input: Power-down
100    RST => '0',                  -- 1-bit input: Reset
101    -- DRP Ports: 7-bit (each) input: Dynamic reconfiguration ports
102    DADDR => (others => '0'),      -- 7-bit input: DRP address
103    DCLK => '0',                  -- 1-bit input: DRP clock
104    DEN => '0',                  -- 1-bit input: DRP enable
105    DI => (others => '0'),         -- 16-bit input: DRP data
106    DWE => '0',                  -- 1-bit input: DRP write enable
107    -- Dynamic Phase Shift Ports: 1-bit (each) input: Ports used for
    => dynamic phase shifting of the outputs
108    PSCLK => '0',                -- 1-bit input: Phase shift clock
109    PSEN => '0',                 -- 1-bit input: Phase shift enable
110    PSINCDEC => '0',             -- 1-bit input: Phase shift
    => increment/decrement
111    -- Feedback Clocks: 1-bit (each) input: Clock feedback ports
112    CLKFBIN => clkfbout
113 );
114
115     clk20 <= clk20_s;            -- Asignar la señal de 20 MHz al puerto
    => de salida
116 end Behavioral;

```

Bibliografía

- [1] AlSharif, M. H., Saad, M., Siala, M., Ballal, T., Boujemaa, H., and Al-Naffouri, T. Y. (2017). Zadoff-chu coded ultrasonic signal for accurate range estimation. In *2017 25th European Signal Processing Conference (EUSIPCO)*, pages 1250–1254.
- [2] Aparicio, J., Álvarez, F. J., Jiménez, A., De Marziani, C., Ureña, J., and Diego, C. (2011). Underwater channel modeling for a relative positioning system. In *OCEANS 2011 IEEE - Spain*, pages 1–7.
- [3] Aparicio, J., Álvarez, F. J., Ureña, J., Jiménez, A., Diego, C., and García, E. (2010). Swell effect in shallow underwater acoustic communications. In *2010 IEEE 15th Conference on Emerging Technologies and Factory Automation (ETFA 2010)*, pages 1–4.
- [4] Aparicio, J., Jiménez, A., Álvarez, F. J., Ureña, J., De Marziani, C., de Diego, D., Cruz, N., and Campos, H. (2015). Accurate detection of spread-spectrum modulated signals in reverberant underwater environments. *Applied Acoustics*, 88:57–65.
- [5] Aparicio, J., Jiménez, A., Álvarez, F. J., Ureña, J., De Marziani, C., and Diego, C. (2013). Influence of different phenomena on the errors in distance measurement using underwater acoustics coded signals. In *2013 MTS/IEEE OCEANS - Bergen*, pages 1–8.
- [6] Association, S. Sd capacity. Disponible en <https://www.sdcard.org/developers/sd-standard-overview/capacity-sd-sdhc-sdxc-sduc/>.
- [7] Association, S. Sd speed class. Disponible en <https://www.sdcard.org/developers/sd-standard-overview/speed-class/>.
- [8] Bai, Y. and Bouvet, P.-J. (2018). Orthogonal chirp division multiplexing for underwater acoustic communication. *Sensors*, 18(11).
- [9] Chu, D. (1972). Polyphase codes with good periodic correlation properties (corresp.). *IEEE Transactions on Information Theory*, 18(4):531–532.
- [10] Digilent (2016a). *Nexys4 DDR™ FPGA Board Reference Manual*. Pullman, WA 99163. Disponible en <https://digilent.com/reference/programmable-logic/nexys-4-ddr/reference-manual>.
- [11] Digilent (2016b). *PmodDA2™ Reference Manual*. Pullman, WA 99163. Disponible en <https://digilent.com/reference/pmod/pmodda2/reference-manual>.
- [12] Frank, R., Zadoff, S., and Heimiller, R. (1962). Phase shift pulse codes with good periodic correlation properties (corresp.). *IRE Transactions on Information Theory*, 8(6):381–382.

- [13] García, E., Ureña, J., and García, J. J. (2013). Generation and correlation architectures of multilevel complementary sets of sequences. *IEEE Transactions on Signal Processing*, 61(24):6333–6343.
- [14] Group, S. (2006). *Physical Layer Simplified Specification Version 2.00*. Disponible en <https://www.sdcard.org/downloads/pls/>, version 2.00.
- [15] Group, S. (2022). *Physical Layer Simplified Specification Version 9.00*. Disponible en <https://www.sdcard.org/downloads/pls/>, version 9.00.
- [16] Instruments, T. (2015). *DAC121S101/-Q1 12-Bit Micro Power, RRO Digital-to-Analog Converter*. Disponible en <https://www.ti.com/lit/ds/symlink/dac121s101.pdf>.
- [17] Kasami, T. (1966). Weight distribution formula for some class of cyclic codes. *Coordinated Science Lab. University of Illinois, Tech. Rep. AD0632574*.
- [18] Khyam, M. O., Xinde, L., Ge, S. S., and Pickering, M. R. (2017). Multiple access chirp-based ultrasonic positioning. *IEEE Transactions on Instrumentation and Measurement*, 66(12):3126–3137.
- [19] Kingston (2010). *microSDHC memory card Flash Storage Media*. Disponible en https://www.kingston.com/datasheets/SDCIT-specsheet-8gb-32gb_en.pdf.
- [20] Klauder, J. R., Price, A., Darlington, S., and Albersheim, W. J. (1960). The theory and design of chirp radars. *Bell System Technical Journal*, page 745–808.
- [21] Liu, H., Wang, X., and Zheng, D. (2018). On the weight distributions of a class of cyclic codes. *Discrete Mathematics*, 341(3):759–771.
- [22] Mow, W. and Li, S.-Y. (1992). Aperiodic autocorrelation properties of perfect polyphase sequences. In *[Proceedings] Singapore ICCS/ISITA '92*, pages 1232–1234 vol.3.
- [23] Murano, S., Pérez, M. C., García, E., Ureña, J., Marziani, C. D., Alcoleas, R., and Colombo, M. (2019a). *Implementación Hardware de un Algoritmo de Correlación Eficiente de Códigos CSS Multinivel*, chapter 6. XXVI SEMINARIO ANUAL DE AUTOMÁTICA, ELECTRÓNICA INDUSTRIAL E INSTRUMENTACIÓN.
- [24] Murano, S., Pérez-Rubio, C., Aparicio, J., and De Marziani, C. (2019b). Doppler resilience evaluation of different encoding techniques for underwater acoustic ranging systems. In *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1627–1630.
- [25] Murano, S., Pérez-Rubio, C., Gualda, D., Álvarez, F. J., Aguilera, T., and Marziani, C. D. (2020). Evaluation of zadoff chu, kasami, and chirp based encoding schemes for acoustic local positioning systems. *IEEE Transactions on Instrumentation and Measurement*, 69(8):5356–5368.
- [26] Murano, S., Pérez-Rubio, M. C., Aparicio, J., Gualda, D., de Vicente-Ranera, J., and Álvaro Hernández (2022). Design of a complete simulator for underwater acoustic localization systems based on spread-spectrum signals. *Applied Acoustics*, 199:109030.

- [27] Selmer, E. S. (1966). *Linear recurrence relations over finite fields*. Department of Mathematics, University of Bergen.
- [28] Van Kleunen, W. A. P., Blom, K. C. H., Meratnia, N., Kokkeler, A. B. J., Havinga, P. J. M., and Smit, G. J. M. (2014). Underwater localization by combining time-of-flight and direction-of-arrival. In *OCEANS 2014 - TAIPEI*, pages 1–6.
- [29] Wang, Y., Qiao, C., and Liu, S. (2016). Design of autonomous underwater vehicle positioning system. In *2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, pages 718–722.