Escuela de Doctorado
y Estudios de Posgrado
Universidad de La Laguna

Máster Universitario en Ingeniería Industrial

Trabajo Fin de Máster

Entrenador de Comunicaciones Digitales basado en FPGA

Autor: Alejandro José Díaz González

Tutor: Oswaldo Bernabé González Hernández

Cotutor: Eduardo Magdaleno Castelló

Septiembre de 2022

La publicación de este Trabajo Fin de Máster solo implica que el estudiante ha obtenido al menos la nota mínima exigida para superar la asignatura correspondiente, no presupone que su contenido sea correcto, aunque si aplicable. En este sentido, la ULL no posee ningún tipo de responsabilidad hacia terceros por la aplicación total o parcial de los resultados obtenidos en este trabajo. También pone en conocimiento del lector que, según la ley de protección intelectual, los resultados son propiedad intelectual del alumno, siempre y cuando se haya procedido a los registros de propiedad intelectual o solicitud de patentes correspondientes con fecha anterior a su publicación.

Resumen

Este trabajo fin de máster aborda el diseño e implementación de un entrenador de comunicaciones digitales sobre la tarjeta de desarrollo *Nexys A7* basada en la FPGA *Artix-7* XC7A100T-CSG324C y la readaptación de otro entrenador de la asignatura Sistemas de Comunicación para su funcionamiento en dicha tarjeta. El proyecto diseñado se puede utilizar para analizar el comportamiento de diferentes técnicas de modulación digital, en concreto, las modulaciones ASK, BPSK, FSK y QPSK, que están presentes en el entrenador.

Se ha utilizado el software *Vivado 2020.1* para realizar el diseño, el cual se compone de varios bloques como un generador de datos pseudoaleatorio, modulador digital, generador de ruido, demodulador digital y, por último, un módulo UART que permite la transferencia de información al ordenador, comunicándose con el software *LabView* el cual permite la visualización de los resultados en tiempo real.

Abstract

This Master's Thesis is focused on the design and implementation of a training system for the study and learning of digital communications systems on the *Nexys A7* development board based on the *Artix-7* XC7A100T-CSG324C FPGA and the re-adaption of another training system of the subject 'Communication Systems' for its operation on this board. The designed project can be used to analyse the behaviour of different digital modulation techniques, specifically ASK, BPSK, FSK and QPSK modulations, which has been implemented in the FPGA board.

The *Vivado 2020.1* software has been used to design the system implemented in the FPGA, which is made up of several blocks such as a pseudo-random data generator, digital modulator, noise generator, digital demodulator and, finally, a UART module that allows for the transfer of information towards the computer, enabling the communication with the *LabView* software which provides the visualisation of the results in real time.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

Capítulo 1. Introducción

Capítulo 1. Introducción

1.1. Introducción general

En el presente proyecto se aborda el Trabajo Fin de Máster correspondiente al Máster de Ingeniería Industrial de la especialidad de Electrónica de la Universidad de La Laguna. A lo largo del documento se expone el desarrollo del diseño y la implementación de un entrenador de comunicaciones digitales sobre la placa de desarrollo *Nexys A7*, que incluye una FPGA[1] XC7A100T-CSG324C de la familia *Artix-7.*

Este Trabajo Fin de Máster surge a raíz de las prácticas de la asignatura de 'Sistemas de Comunicación' del Máster de Ingeniería Industrial donde se analiza el funcionamiento de las diferentes técnicas de modulación digitales básicas estudiadas en la asignatura. En las prácticas, se utiliza una tarjeta de desarrollo basada en una FPGA donde recaían las tareas a ejecutarse, la cual hacía posible la visualización de los resultados en tiempo real en el software *LabView* generadas en la FPGA a través de la transferencia de dicha información utilizando el puerto serie.

Además de realizar el diseño y la implementación de un entrenador de comunicaciones, se pretende adaptar el diseño del entrenador de las prácticas de la asignatura a la nueva tarjeta de desarrollo, la *Nexys A7*.

1.2. Objetivo

El objetivo principal de este Trabajo Fin de Máster es diseñar un sistema de comunicación digital en el que se muestren las diferentes técnicas de modulación digital tanto binarias (ASK[2], BPSK[3] y FSK[4]) como de niveles múltiples (QPSK[5]) con el fin de entender su comportamiento y analizar sus formas de onda. Además, otro de los objetivos es readaptar el código del entrenador de las prácticas de la asignatura 'Sistemas de Comunicación' en otra placa de desarrollo FPGA.

[1] *Field-Programmable Gate Array*
[2] Modulación por desplazamiento en amplitud, de sus siglas en inglés *Amplitude Shift Keying*
[3] Modulación por desplazamiento de fase binaria, de sus siglas en inglés *Binary Phase Shift Keying*
[4] Modulación por desplazamiento de frecuencia, de sus siglas en inglés *Frequency Shift Keying*
[5] Modulación por desplazamiento de fase en cuadratura, de sus siglas en inglés *Quadrature Phase Shift Key*ing

La placa de desarrollo basada en una FPGA lleva a cabo la generalidad de las funciones de los diferentes bloques que componen el sistema, mostrados en la Figura 1, incluido la transferencia de información hacia el ordenador mediante el protocolo UART *(Universal Asynchronous Receiver-Transmitter)*.

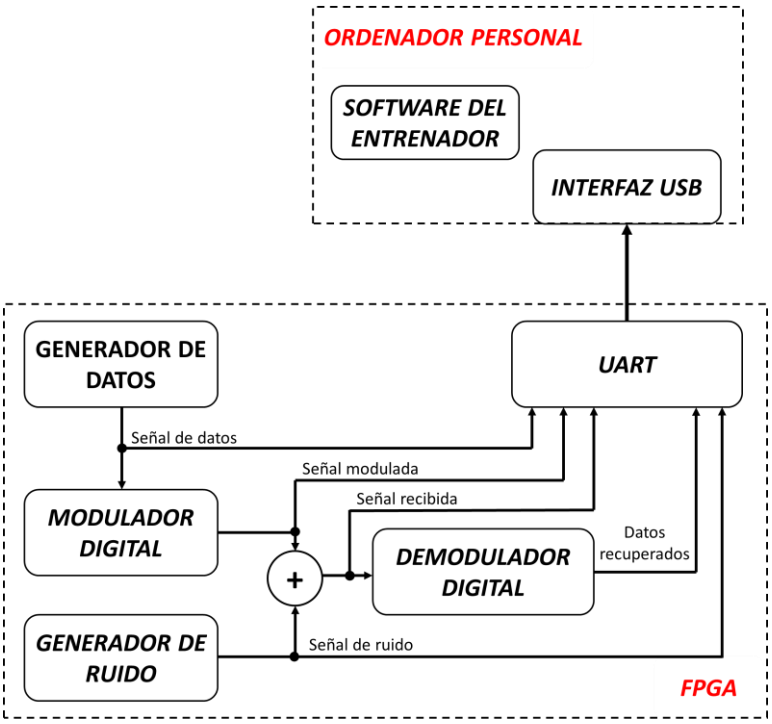


Figura 1. Diagrama de bloques del sistema

En el entrenador de comunicaciones digitales se incluye un transmisor, el cual está compuesto por un generador de datos, ofreciendo secuencias de datos pseudoaleatorias, y un modulador digital, el cual modula la señal en función del dato que reciba. Además, también incluye un receptor, que consta de un demodulador digital, encargado de recuperar la señal original que se desea transmitir. Por otro lado, también está presente un generador de ruido que proporciona el ruido siguiendo una distribución de probabilidad gaussiana y, por último, un módulo UART que permite la transferencia de información entre el ordenador y la placa de desarrollo.

Además, para analizar y observar en tiempo real el comportamiento de las señales de las diferentes técnicas de modulación generadas por el entrenador, se ha desarrollado un software en *LabView* cuyos valores se reciben a través de un puerto serie.

1.3. Estructura de la Memoria

Tras esta introducción, en el capítulo 2 se lleva a cabo una descripción general del sistema donde se describen los aspectos básicos necesarios para poder llevar a cabo la implementación del entrenador. Luego, en el capítulo 3, se realiza una descripción del *Software,* en el cual se describen los diferentes módulos que componen el proyecto, tanto en *Vivado* como en *LabView*, y en el capítulo 4 se profundiza más en el análisis de los códigos y simulaciones, así como en los recursos utilizados. Por consiguiente, en el capítulo 5 se refleja el presupuesto del entrenador, pudiéndose observar los diferentes costes del proyecto. Por último, en los tres últimos capítulos, se plasman la conclusión, bibliografía y los anexos.

Capítulo 2. Descripción general del Sistema

Capítulo 2. Descripción general del sistema

Como la implementación del entrenador de comunicaciones se realiza en una FPGA, se ha optado por realizar su diseño mediante el uso de lenguajes de descripción de hardware de alto nivel, en concreto VHDL *(Very high speed integrated circuit – Hardware Description Language*).

En los siguientes apartados se explicarán las características generales del entrenador de comunicaciones digital.

2.1. Formato de los datos

Dado que dentro de la FPGA se va a estar operando continuamente con datos digitales, se ha establecido el formato de punto fijo para la estructura de estos datos (Figura 2), donde se tiene que el bit más significativo *(*MSB, *Most Significant Bit)* se asigna al signo y el resto conforma el número, teniendo en cuenta que una parte se destina a su parte entera y el resto a su parte decimal.

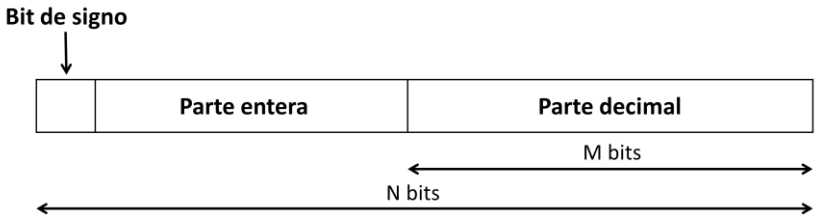


Figura 2. Estructura de los datos en formato de punto fijo

Una de las propiedades de este tipo de formato es que todas las operaciones aritméticas pueden ser calculadas mediante una única operación realizada en aritmética entera y, en el caso de la multiplicación o división, se realiza un desplazamiento, fijo y conocido, de la mantisa que normalice adecuadamente el formato del resultado obtenido [5].

Las operaciones aritméticas que lleva a cabo la FPGA en nuestro proyecto mayoritariamente son la suma, la resta y la multiplicación. Hay que destacar esta última ya que, al realizar el producto de dos datos, el resultado tendrá el doble de bits que los datos (2*N* bits), cuya parte decimal contiene el doble de bits que la parte decimal de operandos (2*M* bits).

Para obtener el resultado en los *N* bits que se están trabajando, siempre que se trabaje con operandos que no produzcan *overflow*, se extraen aquellos bits que se encuentran comprendidos entre la posición *M* y la posición *N* + *M* – 1, como se observa en la Figura 3.

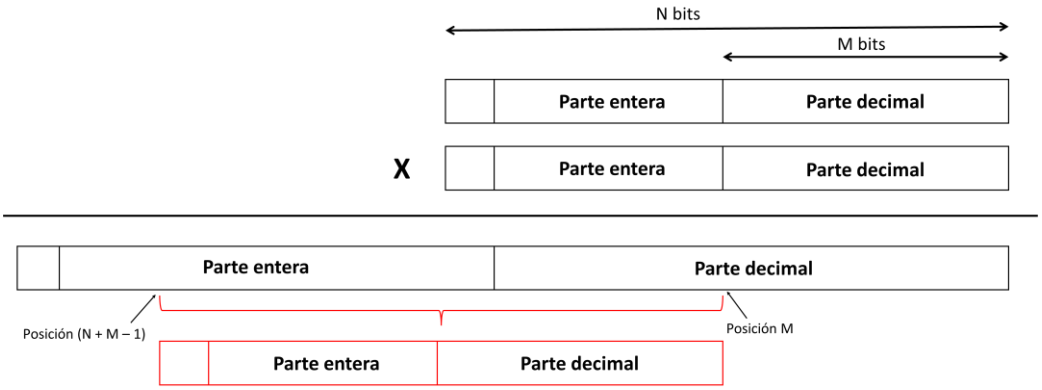


Figura 3. Proceso de multiplicación de dos datos con formato de punto fijo.

Al extraer estos bits, se ha perdido información de la parte decimal[6], los últimos bits que hemos despreciado a la derecha, y por tanto el resultado no es exacto, pero tampoco es un factor crítico a la hora de obtener los resultados.

Además, se trata de una operación de truncamiento, pues simplemente nos quedamos con la parte central del resultado, despreciando los últimos bits de información. Sería un poco más preciso realizar una operación de redondeo, pero resulta mucho más costoso de implementar en lenguaje VHDL y la ganancia en precisión sigue sin ser significativa.

2.2. Generación de las portadoras

Para generar las señales portadoras necesarias en el modulador y en el demodulador, se utiliza una tabla de *K* posiciones que contiene los valores de la señal senoidal y, de manera simultánea, dos punteros desfasados 90º uno respecto al otro, o lo que es lo mismo, desfasados *K*/4 posiciones para así obtener por un lado la onda seno y, por otro lado, la onda coseno. También se tendrán que generar otras señales desfasadas unos ciertos ángulos, necesarias para la modulación QPSK.

[6] Obsérvese que en la parte entera se supone que no perdemos datos, dado que no se ha producido desbordamiento (*overflow*) en la operación de multiplicación.

Los valores de la tabla vienen dados por la posición en la que se encuentre y el número total de posiciones siguiendo la ecuación (1). En nuestro caso, se ha decidido utilizar una tabla de 32 posiciones.

$$Valor\ tabla = sen\left(\frac{2\pi}{Nº\ total\ de\ posiciones} \cdot Posición\ actual\right), \qquad (1)$$

Estos punteros incrementarán su posición en una unidad por cada flanco de la señal de reloj, de forma que se irá leyendo el valor de la tabla al que esté apuntando y, de esta manera, se obtendrán las dos formas de onda. Cuando el puntero llega a la última posición, en el siguiente flanco de reloj pasará a regresar a la posición inicial.

Para verlo de una forma más sencilla, se supone una tabla de 32 posiciones (obsérvese la Figura 4), de manera que la posición inicial del puntero que dibuja la onda coseno será la octava posición. Cuando llegue a la posición 31, regresará a la posición 0 en el siguiente flanco de reloj, dibujando de nuevo la onda.

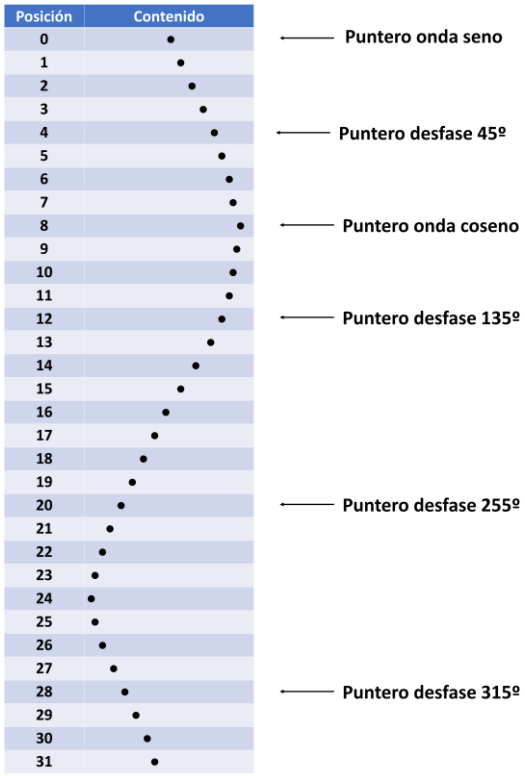


Figura 4. Generación de las señales portadoras mediante punteros

Cabe destacar que la onda coseno únicamente será necesaria para modulaciones con componentes en cuadratura (QPSK) o por el demodulador. Por

otro lado, para la modulación FSK, en lugar de incrementar en una unidad los desplazamientos a los punteros en cada ciclo de reloj, se aumentarán en valores diferentes a la unidad, de manera que, si se aumenta en dos, la onda generada tendrá el doble de frecuencia que la señal de referencia definida por la tabla, es decir, la frecuencia de la señal será múltiplo del desplazamiento de los punteros.

2.3. Generador de datos

El generador de datos se basa en una estructura FSR *(Feedback Shift Register)* que consta de una serie de biestables tipo D donde ciertas salidas de los mismos están realimentadas, utilizando determinadas combinaciones de realimentación a través de una puerta XOR, con la finalidad de ofrecer una secuencia pseudoaleatoria de máxima longitud. La combinación de realimentación viene dada por una teoría matemática compleja denominada “Teoría de Campos Finitos de Galois” [2], la cual hace posible obtener, a la salida del último biestable, una secuencia pseudoaleatoria de valores con una periodicidad de $2^R - 1$ bits, donde R es el número de registros.

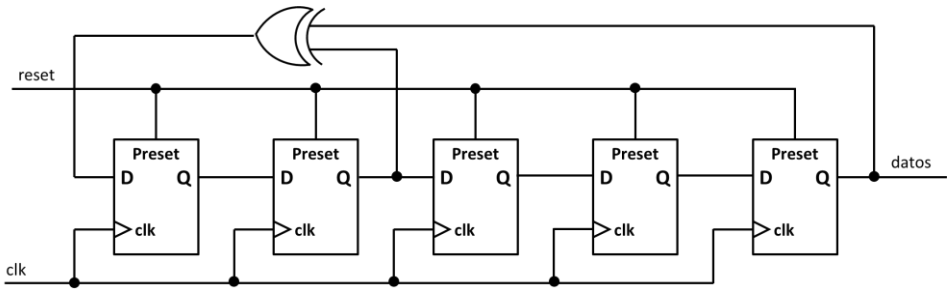


Figura 5. Estructura del generador de datos

En la Figura 5 se muestra la estructura para el generador pseudoaleatorio de cinco registros implementado en el entrenador, donde es necesario que las salidas del segundo y último registro se realimenten a través de una puerta XOR para obtener una secuencia de longitud máxima. Así, se han tomado los n bits menos significativos *(*LSB, *Least Significant Bits*) para disponer del dato aleatorio a utilizar en cada caso por el modulador, donde $n = 1$ para modulaciones binarias y $n > 1$ para modulaciones de niveles múltiples, como QPSK donde $n = 2$.

Los biestables deben inicializarse a nivel lógico ‘1’, ya que, por el contrario, si se inicializan todos a un valor ‘0’, provocaría una salida a nivel lógico bajo de manera indefinida.

2.4. Generador de ruido

Se ha obtenido la señal de ruido AWGN *(Additive White Gaussian Noise)* a partir de una tabla con 256 valores que contenían las muestras que configuraban la función de distribución de probabilidad de un proceso aleatorio gaussiano. Además, se ha utilizado un generador pseudoaleatorio de 21 registros, de manera que su periodicidad se corresponda a 2.097.151 ciclos para hacerlo lo más aleatorio posible.

Así, utilizando los FSR, se toma la salida simultánea de los ocho últimos registros para obtener un número que se distribuya uniformemente entre 0 y 255, la cual se compara con los distintos valores de la tabla hasta encontrar aquél más próximo.

La amplitud del ruido aleatorio se corresponde con la posición en la tabla de dicho valor. Cabe destacar que el usuario puede configurar la ganancia de la amplitud del ruido mediante los *switches* de la FPGA.

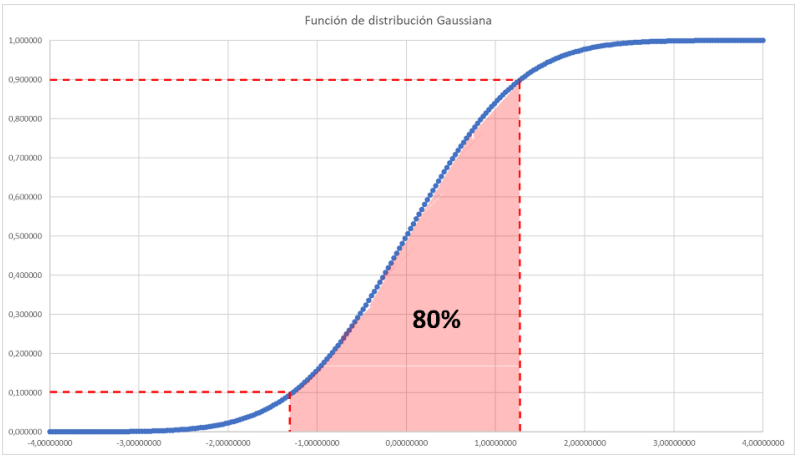


Figura 6. Función de distribución de probabilidad Gaussiana

Obsérvese en la Figura 6 que, si nos desplazamos de manera uniforme en el eje vertical, lo que va a ocurrir es que va a haber mayor probabilidad de obtener amplitudes pequeñas, ya que nos estamos moviendo en el 80% de los casos en estas amplitudes, mientras que, en amplitudes mayores, la probabilidad de ocurrencia es mucho menor.

2.5. Modulador digital

En el modulador digital, dos portadoras en cuadratura se modulan en amplitud por las componentes en fase x_i y en cuadratura y_i del símbolo *i*-ésimo a transmitir,

obtenidos de la constelación correspondiente a partir de la palabra pseudoaleatoria de *n* bits suministrada por el generador de datos. En la Figura 7 se muestra la estructura del modulador.

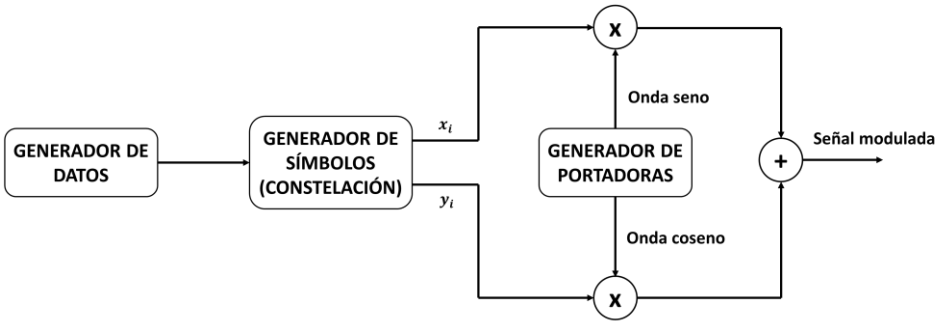


Figura 7. Estructura del modulador

La señal modulada viene definida por la ecuación (2).

$$s_i(t) = x_i \sin(\omega_c t) + y_i \cos(\omega_c t), \qquad iT < t < (i+1)T, \tag{2}$$

Donde $f_c = \omega_c / 2\pi$ es la frecuencia de la señal portadora definida por el número *K* de muestras de la tabla de la portadora de onda seno y la frecuencia del reloj del puntero y *T* es el tiempo de símbolo. Para señales binarias, donde *n* = 1 (BPSK, FSK y ASK), los símbolos no tienen componente en cuadratura ($y_i = 0$). El caso FSK, aun siendo también una modulación binaria, es distinto, puesto que aquí se obtiene una señal de salida a dos frecuencias distintas, en función del dato a transmitir.

2.6. Demodulador de datos

El demodulador se basa en la estructura del detector óptimo frente al ruido AWGN (Figura 8) donde la señal recibida es multiplicada por las portadoras en fase y en cuadratura[7]. Luego, se integra utilizando dos integradores de tal forma que se obtiene una estimación de la parte real e imaginaria del símbolo recibido. A la salida,

[7] En el caso FSK, la rama en cuadratura se sustituye por la multiplicación por la segunda portadora (a frecuencia distinta, múltiplo de la frecuencia de la portadora base), para obtener la correlación entre la señal de entrada y ésta. Dado que dos señales a frecuencias muy distintas, en este caso una múltiplo de la otra, son ortogonales, sólo en la rama en que coincida la frecuencia de la portadora de referencia con la de la señal de entrada, se obtendrá un valor significativo a la salida del integrador y podemos saber qué dato fue transmitido en origen.

el decisor debe determinar el símbolo de la constelación que más se aproxime al símbolo demodulado. El decisor, en el caso binario, no es más que un comparador donde el umbral se establece en el punto medio de los dos posibles símbolos recibidos. En el caso de la demodulación de señales QPSK, se utilizan dos comparadores con los umbrales establecidos en el punto medio y, al ser dos comparadores, habría dos situaciones por cada uno de ellos, es decir, cuatro casos posibles en total.

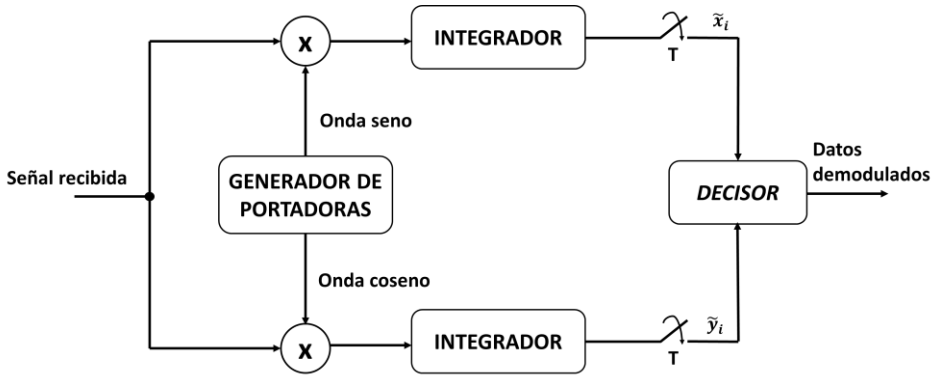


Figura 8. Estructura del demodulador (detección óptima)

Capítulo 3. Diseño del entrenador de comunicaciones

Capítulo 3. Diseño del entrenador de comunicaciones

3.1. Diseño en Vivado

Como se pudo observar en la Figura 1, las funciones de los elementos principales de los sistemas de comunicación como son el transmisor, el medio de transmisión y el receptor, se realizan en la FPGA. El sistema ha sido diseñado en el entorno *Vivado*, cuyos datos son intercambiados mediante un módulo UART hacia el ordenador personal, el cual se comunica con el programa *LabView* donde se visualizan los datos en tiempo real.

En el anexo 8.1 se muestra el diagrama de bloques del diseño en *Vivado* donde se muestran los diferentes bloques principales que componen el sistema.

3.1.1. Módulo Top Level

Este módulo se encarga de instanciar y realizar las conexiones de todos los componentes del sistema, además de seleccionar el tipo de modulación a través del dato que se ha recibido a través del módulo de recepción de la UART. Los puertos de entrada y salida de este módulo se observan en la Tabla 1 y el diagrama de bloques en la Figura 9 (para algunas señales, se han utilizado colores como indicativo de unión para mejorar la visualización del esquema).

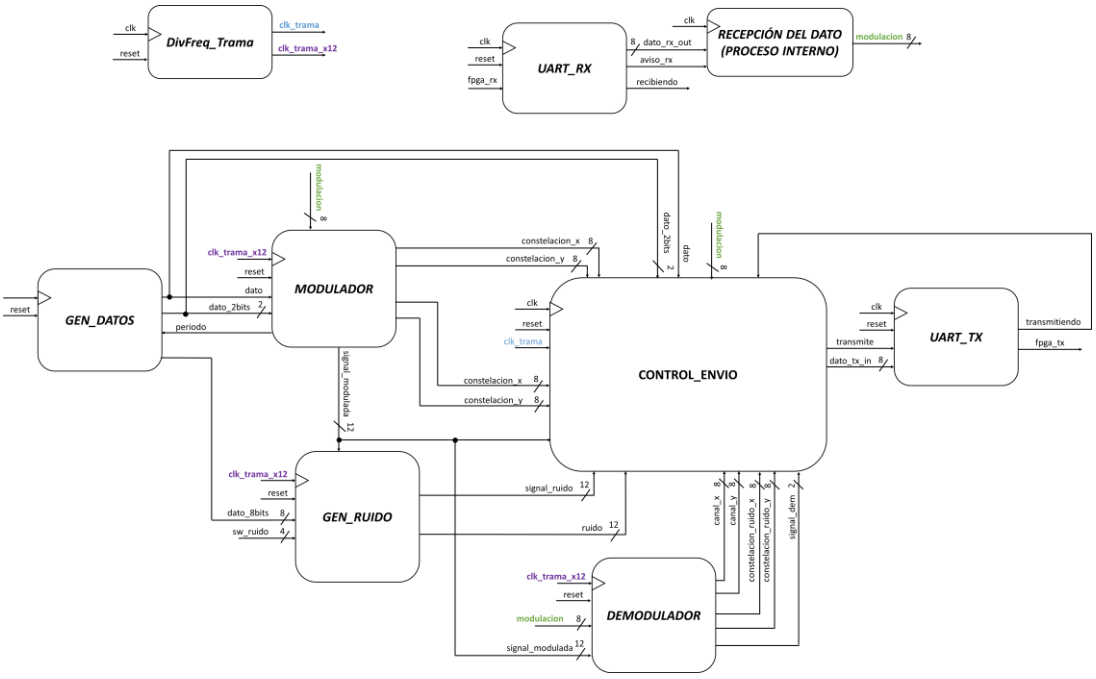


Figura 9. Diagrama de bloques del módulo Top level

Puerto	Bits	I/O	Descripción
clk	1	I	Señal de reloj de la placa
reset	1	I	Señal de *reset* asíncrono
sw_ruido	4	I	Señal de los *switches* de la placa que controlan la ganancia del ruido
fpga_tx	1	O	Trama que se envía en serie al ordenador
fpga_rx	1	I	Trama que se recibe en serie desde el ordenador

Tabla 1. Puertos de entrada y salida del módulo del Top level

Los módulos del modulador, generador de datos, demodulador y generador de ruido deben cambiar sus salidas después de que se hayan enviado los doce paquetes por la UART, por tanto, la señal de reloj que controla estos módulos debe ser la señal periódica **clk_trama_x12**.

Además, en este módulo se encuentra un proceso que guarda el valor del dato recibido a través de la UART que indica el tipo de modulación seleccionado por el usuario cuando la señal **aviso_rx** se encuentra activa.

3.1.2. Módulo UART

Para establecer la comunicación de la FPGA con el puerto serie del ordenador, se ha realizado el diseño de un transmisor/receptor serie asíncrono (UART) el cual permite la recepción y transmisión de datos de manera simultánea (*full-duplex*). Los puertos de la placa que permiten la comunicación son: la línea de transmisión de datos (*Tx*) y la de recepción (*Rx*). Desde el punto de vista de la FPGA, la señal *Rx* es la que se transmite al ordenador, mientras que la señal *Tx* es la que se recibe del ordenador. La placa cuenta con más pines que están incluidos en el protocolo, sin embargo, sólo se utilizaron los dos imprescindibles [4].

Cabe destacar que el diseño se ha realizado con el fin de que sea configurable, es decir que, si en algún momento se desea cambiar la tasa de baudios de la UART, o se utiliza una placa con una frecuencia distinta, no se tenga que rehacer todo el diseño, sino que se cambien el valor de las constantes guardadas en la librería *"UART_PKG"*.

3.1.2.1. Módulo del transmisor

Este módulo se encarga de transmitir los datos organizados en tramas de 8 bits al ordenador mediante la línea de transmisión de datos. Cuando esta línea permanece a nivel alto, no se envían datos. Para empezar la transmisión, el

transmisor debe enviar un bit de inicio poniendo la línea a cero. Luego, se envían los bits de datos de forma consecutiva, empezando por el bit menos significativo (LSB). Al terminar de enviar el último bit de dato, se cierra la trama con un bit de fin poniendo la línea a nivel alto [4]. Todo esto se puede observar en la Figura 10 donde se muestra el cronograma de envío de una trama.

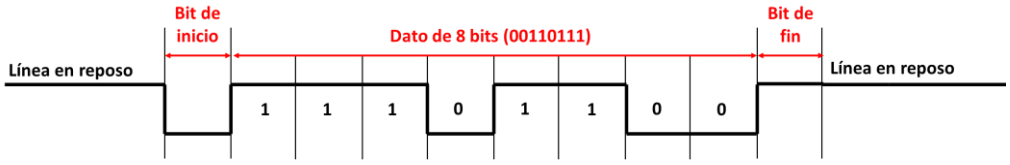


Figura 10. Cronograma de envío de una trama

Las entradas y salidas de este módulo se muestran en la Tabla 2.

Puerto	Bits	I/O	Descripción
clk	1	I	Señal de reloj de la placa
reset	1	I	Señal de *reset* asíncrono
transmite	1	I	Señal del sistema que ordena al módulo la transmisión del dato que se encuentra en **dato_tx_in**. La orden dura un único ciclo de reloj
dato_tx_in	8	I	Dato que se quiere enviar, se proporciona cuando se activa **transmite** (**transmite** = '1')
transmitiendo	1	O	Indica al sistema que el módulo está transmitiendo y, por tanto, no podrá atender a ninguna nueva orden de transmisión. Ignorará a **transmite** cuando **transmitiendo** se encuentre a nivel alto
fpga_tx	1	O	Trama que se envía en serie al ordenador

Tabla 2. Puertos de entrada y salida del módulo de transmisión de la UART

El diagrama de bloques del módulo del transmisor se muestra en la Figura 11 y consta de los siguientes elementos:

- Un divisor de frecuencia para generar una señal periódica que indica cuándo ha pasado el intervalo de tiempo correspondiente a cada bit de la trama de envío con la frecuencia indicada en los baudios. En este caso, se optó por una frecuencia de 115 200 bits/s.
- Un registro de carga paralela (*parallel-in serial-out:* PISO) que guarda el dato que se desea enviar, y que lo desplaza según el bit que se esté enviando.
- Un multiplexor, que envía el bit correspondiente según el estado en el que se encuentre.

- Un bloque de control que indica al resto de bloques en qué estado se encuentra, es decir, lo que corresponde enviar.

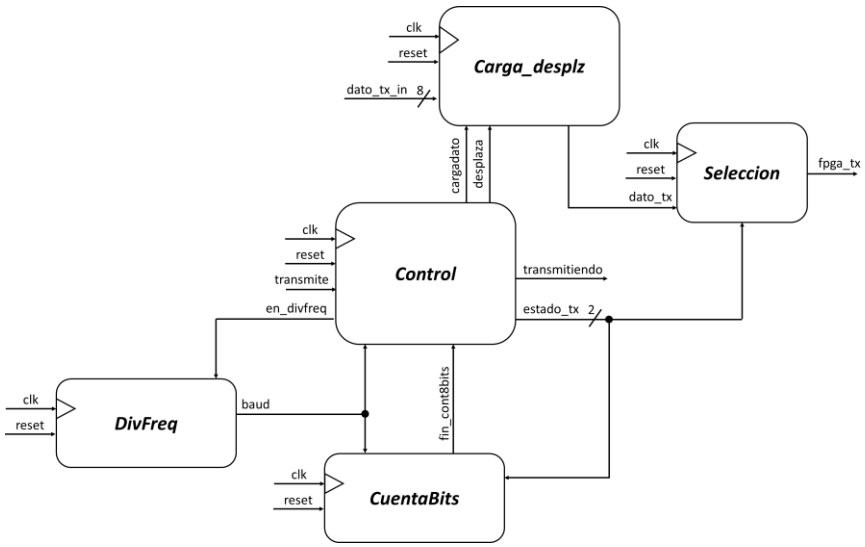


Figura 11. Diagrama de bloques del transmisor UART.

Para realizar el control del módulo del transmisor, se ha realizado un diagrama de estados con cuatro estados (véase la Figura 12): uno que indica el estado de reposo *(***e_init***)*, el envío del bit de inicio *(***e_bit_init***)*, el envío de los ocho bits de datos *(***e_bits_dato***)* y, por último, el bit de fin *(***e_bit_fin***)*.

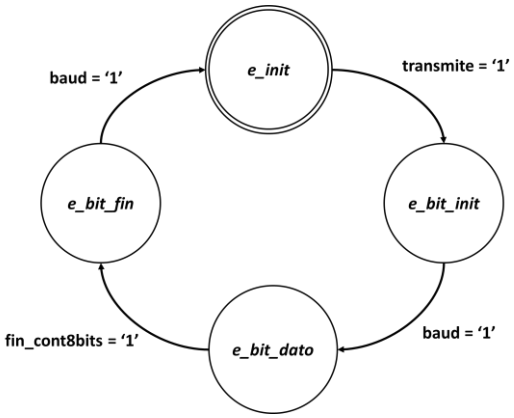


Figura 12. Diagrama de estados del transmisor de la UART.

Las funciones que deben desempeñar cada uno de los estados son:

- **Estado inicial:** En este estado, el transmisor se encuentra esperando a la orden de transmitir, por tanto, la línea de transmisión estará activa a alta (**fpga_tx = '1'**). En el momento en el que se quiera transmitir un dato, la señal **transmite** se encontrará a '1' y por consiguiente se empezará a enviar los bits

empezando por el bit de inicio. En ese momento, se pone la señal **cargadato** a '1' para guardar el dato que se quiere enviar en el registro de desplazamiento. Además, el contador del divisor de frecuencia debe estar deshabilitado en este estado ya que no se está enviando ningún bit por el momento.

- **Envío del bit de inicio:** En este estado, se pone la línea de transmisión a baja (**fpga_tx = '0'**) indicando el envío del bit de inicio. Cuando haya pasado el tiempo correspondiente del envío de un bit, se pasará al siguiente estado.
- **Envío de los bits de datos:** En este estado se envían los ocho bits de datos. Se ha implementado un contador que cuente el número de bits que se están enviando, de manera que cuando llegue al final de la cuenta, se active una señal que lo indique (**fin_cont8bits**) y se cambie de estado. Además, la señal **desplaza** indicará al registro de desplazamiento que tiene que desplazar sus bits, determinado por el tiempo de envío generado por el divisor de frecuencia.
- **Envío del bit de fin:** En este estado, se pone la línea de transmisión a alta (**fpga_tx = '1'**) indicando el envío del bit de fin. Se saldrá de este estado cuando haya pasado el periodo correspondiente a un bit.

En la Tabla 3 se muestra la tabla de estados, entradas y salidas.

Estado actual	Entradas			Estado siguiente	Salidas			
	transmite	baud	fin_cont8bits		cargadato	desplaza	en_divfreq	transmitiendo
e_init	0	X	X	e_init	0	0	0	0
e_init	1	X	X	e_bit_init	1	0	0	0
e_bit_init	X	0	X	e_bit_init	0	0	1	1
e_bit_init	X	1	X	e_bits_dato	0	0	1	1
e_bits_dato	X	X	0	e_bits_dato	0	baud	1	1
e_bits_dato	X	X	1	e_bit_fin	0	0	1	1
e_bit_fin	X	0	X	e_bit_fin	0	0	1	1
e_bit_fin	X	1	X	e_init	0	0	1	1

Tabla 3. Tabla de estados, entradas y salidas del transmisor de la UART.

3.1.2.2. Módulo del receptor

Este módulo se encarga de recibir los datos organizados en tramas de 8 bits del ordenador mediante la línea de recepción de datos. La estructura de la trama es la misma que para el transmisor ya mostrada en la Figura 10.

Las entradas y salidas de este módulo se muestran en la Tabla 4.

Puerto	Bits	I/O	Descripción
clk	1	I	Señal de reloj de la placa
reset	1	I	Señal de *reset* asíncrono
fpga_rx	1	I	Trama que se recibe en serie desde el ordenador.
aviso_rx	1	O	Aviso de que se ha recibido un nuevo dato y que está disponible en **dato_rx_out**. El aviso se da poniendo la señal a '1' durante un único ciclo de reloj
dato_rx_out	8	O	Proporciona los ocho bits del dato que se ha recibido. Este dato sólo será válido desde que **aviso_rx** valga '1' y mientras **recibiendo** sea '0'
recibiendo	1	O	Cuando vale '1' indica que el módulo se encuentra recibiendo una trama y por tanto el valor del dato **dato_rx_out** no es válido

Tabla 4. Puertos de entrada y salida del módulo de recepción de la UART

El diagrama de bloques del módulo del receptor se muestra en la Figura 13 y consta de los siguientes elementos:

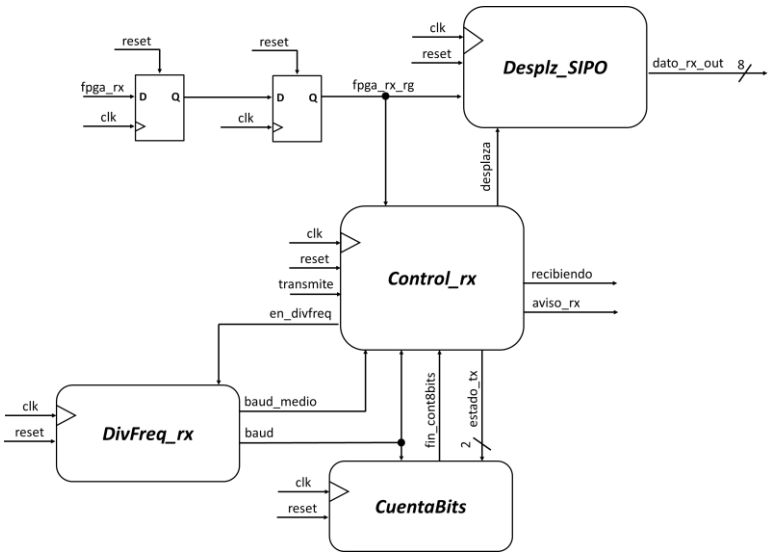


Figura 13. Diagrama de bloques del receptor

- Un registro de desplazamiento al que se le van cargando los datos en serie y los devuelve en paralelo (*serial-in parallel-out:* SIPO), es decir, el opuesto al que se usó en el transmisor.
- Un registro de entrada para registrar la señal **fpga_rx** debido a que es asíncrona y para evitar meta-estabilidad[8].
- Un divisor de frecuencia que, a diferencia del transmisor, debe sincronizarse con la trama que se recibe ya que el receptor no es el que dirige la transmisión. Posee una señal de habilitación **en_divfreq** la cual hace que el divisor se mantenga inactivo hasta que el valor de esta señal sea '1'. El divisor tiene dos salidas, una que marca la transición del estado y otra el punto medio de cada bit de la transmisión (**baud_medio**). En este punto medio es donde se evalúa el valor del bit recibido, evitando evaluar el bit cerca de las transiciones donde se puede tomar el valor del bit contiguo.
- Un bloque de control que indica al resto de bloques en qué estado se encuentra, similar al del transmisor.

Básicamente, el control del módulo de recepción es muy similar al módulo de transmisión, posee los cuatro mismos estados, pero con la diferencia de que posee señales distintas que indican el paso de un estado a otro. Además, la señal indicadora de recepción de datos es la propia línea de recepción **fpga_rx**, la cual, cuando pasa de estar en el estado de reposo en nivel bajo a nivel alto, es cuando pasa al siguiente estado de envío del bit de inicio. Esto se muestra en la Figura 14 donde se observa el diagrama de estado del módulo de recepción y en la Tabla 5 donde se muestra la tabla de estados, entradas y salidas de este módulo.

Cabe destacar que el dato recibido sólo será válido a partir del aviso de la señal **aviso_rx** y mientras que la señal **recibiendo** se mantenga a '0'. Estas señales nunca podrán estar activas simultáneamente. En el momento en el que **recibiendo** se encuentre en alta, el dato no será válido.

[8] La metaestabilidad es un estado inestable de un biestable, en el que no se encuentra a un valor '0' o un valor '1'. Este estado se produce por no cumplirse los parámetros temporales del biestable, es decir, cuando no se respetan los tiempos de *setup* y *hold* [3].

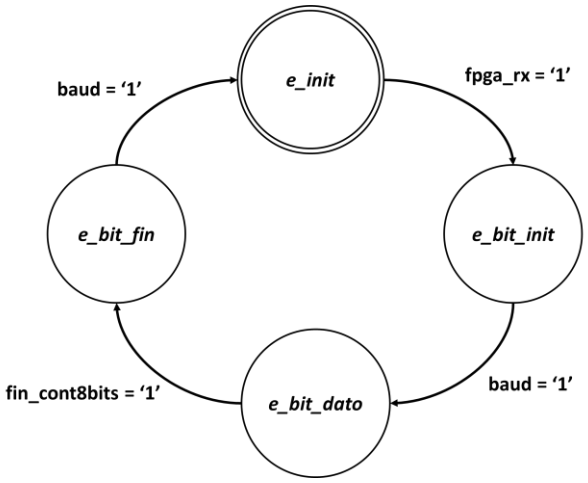


Figura 14. Diagrama de estados del receptor de la UART.

Estado actual	Entradas				Estado siguiente	Salidas			
	fpga_rx	baud_medio	baud	fin_cont8bits		en_divfreq	recibiendo	aviso_rx	desplaza
e_init	1	X	X	X	e_init	0	0	0	0
e_init	0	X	X	X	e_bit_init	0	0	0	0
e_bit_init	X	1	0	X	e_bit_init	1	1	0	baud_medio
e_bit_init	X	X	1	X	e_bits_dato	1	1	0	0
e_bits_dato	X	X	X	0	e_bits_dato	1	1	0	baud_medio
e_bits_dato	X	X	X	1	e_bit_fin	1	1	0	0
e_bit_fin	X	X	0	X	e_bit_fin	1	1	0	0
e_bit_fin	X	X	1	X	e_init	1	0	1	0

Tabla 5. Tabla de estados, entradas y salidas del receptor de la UART.

3.1.3. Módulo del generador de datos

Este módulo se encarga de generar los datos pseudoaleatorios tanto para la generación del ruido como para la modulación de las señales. En este bloque es donde se crea la estructura de los FSR, instanciando *R* veces los registros en cada caso, utilizando bucles *for*.

Los puertos de entrada y salida de este bloque se muestran en la Tabla 6.

Puerto	Bits	I/O	Descripción
clk	1	I	Señal de reloj de la placa
clk_periodo	1	I	Señal periódica que marca los ciclos que tiene la señal en un bit de dato
reset	1	I	Señal de *reset* asíncrono
dato	1	O	Dato pseudoaleatorio de 1 bit para las modulaciones ASK, BPSK y FSK
dato_2bits	2	O	Dato pseudoaleatorio de 2 bits para la modulación QPSK
dato_8bits	8	O	Dato pseudoaleatorio de 8 bits para la generación de ruido

Tabla 6. Puertos de entrada y salida del módulo del generador de datos

Además, también se encuentran dos constantes declaradas en la entidad (Tabla 7).

Parámetro	Bits	Tipo	Valor	Descripción
Nreg	1	Positivo	5	Número de registros para el generador de datos usado para las diferentes modulaciones
Nreg_8	1	Positivo	21	Número de registros para el generador de datos de 8 bits para la generación de ruido

Tabla 7. Constantes del módulo del generador de datos

La señal de reloj del generador de datos de cinco registros indica cuándo se genera un nuevo dato a la salida del generador. Se ha decidido que la generación de cada nuevo dato se corresponda con dos ciclos de la señal portadora, es decir, el *tiempo de símbolo*[9] es equivalente a dos ciclos de la señal portadora.

3.1.4. Módulo del modulador

Este módulo es el encargado de generar la señal modulada en función del tipo de modulación que haya escogido el usuario (recibido desde el módulo de recepción de la UART).

En este módulo se encuentra un contador encargado de generar la señal periódica que marca los ciclos que tiene la señal en un bit de dato. Como se ha decidido que un símbolo de datos (un bit en el caso de los sistemas binarios) tenga dos ciclos de la señal senoidal y la tabla de generación de las portadoras tiene 32 posiciones, el periodo completo se corresponde a 64 cuentas del contador.

Los puertos de entrada y salida de este módulo se muestran en la Tabla 8.

[9] El tiempo de símbolo constituye la duración en segundos del símbolo digital a transmitir y es igual a la inversa de la tasa de símbolo en símbolos por segundo, también conocido como baudios. En el caso de los sistemas de comunicación digitales binarios, el tiempo de símbolo se corresponde con el tiempo de bit y su inversa es la tasa de transmisión en bits por segundo.

Puerto	Bits	I/O	Descripción
clk	1	I	Señal de reloj de la placa
reset	1	I	Señal de *reset* asíncrono
dato	1	I	Dato pseudoaleatorio de 1 bit para las modulaciones ASK, BPSK y FSK
dato_2bits	2	I	Dato pseudoaleatorio de 2 bits para la modulación QPSK
modulacion	8	I	Señal que indica el tipo de modulación seleccionado
signal_modulada	12	O	Señal modulada
contelacion_x	8	O	Constelación del canal *x* (fase) sin ruido
constelación_y	8	O	Constelación del canal *y* (cuadratura) sin ruido
periodo	1	O	Señal periódica que marca los ciclos que tiene la señal en un bit de dato

Tabla 8. Puertos de entrada y salida del módulo del modulador

Por otro lado, hay otro proceso que se encarga de generar las portadoras para cada modulación y, en función del tipo que haya seleccionado el usuario y del valor del dato pseudoaleatorio, se irá construyendo la señal modulada y guardando el valor de las constelaciones.

Por ejemplo, para el caso de la modulación ASK, si el dato está a '1', la señal modulada debe tener la forma de la onda seno hasta que el dato se encuentre a '0', que es cuando la señal toma un valor constante e igual a cero. Para el caso de las modulaciones BPSK y FSK, ocurre lo mismo cuando el dato se encuentra en '1' pero con la diferencia de que cuando el dato esté a '0', la señal debe seguir la forma de la señal seno negativa para el caso de BPSK, y la señal seno de doble frecuencia para el caso de la FSK. Para el caso de la modulación QPSK, se ha utilizado *Codificación Gray*[10] de manera que cuando el dato de dos bits sea "00", la señal modulada siga la forma de la onda seno desfasada 45º, cuando sea "01" siga la forma de la onda seno desfasado 135º, si es "11" siga la forma de la onda seno desfasada 255º y, por último, si es "10", siga la forma de la onda seno desfasada 315º.

Para la generación de las señales desfasadas, se ha colocado el puntero en las posiciones iniciales que se muestran en la Figura 4.

3.1.5. Módulo del generador de ruido

Este módulo se encarga de generar la amplitud de ruido y de adición del ruido. Sus puertos de entrada y salida se muestran en la Tabla 9.

[10] En codificación Gray, sólo cambia un bit entre dos símbolos adyacentes. Para el caso de QPSK (dos bits), se tendría que los símbolos adyacentes son 00 → 01 → 11 → 10 → 00.

Puerto	Bits	I/O	Descripción
clk	1	I	Señal de reloj de la placa
reset	1	I	Señal de *reset* asíncrono
sw_ruido	4	I	Señal de los *switches* de la placa que controlan la ganancia del ruido
dato_8bits	8	I	Dato pseudoaleatorio de 8 bits para la generación del ruido
signal_modulada	12	I	Señal modulada (sin ruido)
signal_ruido	12	O	Señal modulada con ruido
amplitud_ruido	12	O	Amplitud de ruido obtenida de la tabla de la distribución normal

Tabla 9. Puertos de entrada y salida del módulo del generador de ruido

Este módulo consta principalmente de dos procesos: el primero de ellos se encarga de extraer la amplitud del ruido de la tabla de distribución normal que se encuentra en una de las librerías, mientras el otro se encarga de sumar este ruido a la señal modulada con la ganancia correspondiente en función de los *switches* activados de la FPGA.

3.1.6. Módulo del demodulador

Este módulo se encarga de demodular la señal modulada, es decir, recuperar la información de los datos extrayendo la información de las ondas portadoras. Sus puertos de entrada y salida se muestran en la Tabla 10.

Puerto	Bits	I/O	Descripción
clk	1	I	Señal de reloj de la placa
reset	1	I	Señal de *reset* asíncrono
modulacion	8	I	Señal que indica el tipo de modulación seleccionado
signal_ruido	12	I	Señal modulada con ruido
canal_x	8	O	Señal a la salida del integrador en fase
canal_y	8	O	Señal a la salida del integrador en cuadratura
constelacion_ruido_x	8	O	Señal que contiene la constelación resultante al integrar la señal con ruido del canal *x*
constelacion_ruido_y	8	O	Señal que contiene la constelación resultante al integrar la señal con ruido del canal *y*
signal_dem	2	O	Señal con los datos demodulados

Tabla 10. Puertos de entrada y salida del módulo del demodulador

En este módulo, se obtienen las portadoras según el caso. Para la modulación FSK, la portadora del canal *x* se corresponde a la onda seno mientras que la portadora del canal *y* se corresponde a la onda seno de doble frecuencia. Por el contrario, para el resto de las modulaciones, las ondas portadoras serán seno y coseno para los canales *x* e *y* respectivamente.

Para demodular la señal, como se observó en la Figura 8, se multiplica la señal modulada por las portadoras y luego se integran. El problema de esto es que estas operaciones no se realizan en el mismo instante de tiempo, por lo que se introducen retardos. Si la señal **cont_periodo** fuese la que controlase la multiplicación, la suma acumulativa y el guardado del último valor del integrador para obtener las constelaciones, se perderían datos. Por ello, se ha registrado esta señal tres veces, solucionando estos retardos y obteniendo todos los datos de las operaciones.

Una vez integrada la señal, ésta entra al bloque decisor en el cual se han establecido los valores umbrales para cada modulación. Estos valores umbrales se han establecido demodulando la señal sin ruido y observando en la simulación cuál es el valor medio de la señal a la salida del integrador. Además, antes de resetear el integrador al llegar al tiempo de bit, se guarda el último valor que toma la señal a la salida del integrador para obtener así las constelaciones de cada canal.

3.1.7. Módulo del divisor de frecuencia de la trama

Este módulo se encarga de generar dos señales periódicas cuyos periodos se corresponden a lo que tarda en enviarse una trama y otro a lo que tardan en enviarse 12 de estas tramas.

Los puertos de entrada y salida de este módulo se muestran en la Tabla 11.

Puerto	Bits	I/O	Descripción
clk	1	I	Señal de reloj de la placa
reset	1	I	Señal de *reset* asíncrono
clk_trama	1	O	Señal periódica que indica lo que tarda en enviarse una trama
clk_trama_x12	1	O	Señal periódica que indica lo que tardan en enviarse doce tramas

Tabla 11. Puertos de entrada y salida del módulo del divisor de frecuencia de la trama

Este módulo es bastante sencillo ya que está compuesto por dos divisores de frecuencia. Para el diseño del divisor de frecuencia en el que se obtiene la señal periódica que indica lo que tarda en enviarse una trama, primero se obtuvo la cuenta mínima necesaria para enviar una sola trama.

$$Cuenta\ mínima = \frac{Reloj\ de\ la\ placa}{\frac{Tasa\ de\ baudios}{Bits\ enviados\ en\ una\ trama}} = \frac{100 \cdot 10^6\ \text{Hz}}{\frac{115200\ \text{bit/s}}{10\ \text{bits}}} = 8680\ \text{cuentas}$$

Al introducir esta cuenta, se observó que se introducían errores de envío, por lo que se decidió aumentar mucho más el valor final de la cuenta para solventar este error, hasta 50 000 cuentas.

En cuanto al otro divisor de frecuencia, el valor de la cuenta se estableció en el número de paquetes a enviar, siendo la señal de reloj que controlaba este elemento la señal periódica obtenida en el anterior divisor. La señal resultante se utiliza para ir actualizando los valores de las señales del resto de módulos, con el fin de que no se pierda ninguna información.

3.1.8. Módulo del control del envío

Este módulo se encarga de adecuar las señales para su envío por la UART y organizar el orden de envío de los paquetes.

Puerto	Bits	I/O	Descripción
clk	1	I	Señal de reloj de la placa
reset	1	I	Señal de *reset* asíncrono
modulacion	8	I	Señal que indica el tipo de modulación seleccionado
dato	1	I	Dato pseudoaleatorio de 1 bit
dato_2bits	2	I	Dato pseudoaleatorio de 2 bits
transmitiendo	1	I	Indica al sistema que el módulo está transmitiendo
signal_modulada	12	I	Señal modulada (sin ruido)
signal_ruido	12	I	Señal modulada con ruido
ruido	12	I	Amplitud de ruido obtenida de la tabla de la distribución normal
clk_trama	1	I	Señal periódica que indica lo que tarda en enviarse una trama
signal_dem	2	I	Señal con los datos demodulados
canal_x	8	I	Señal a la salida del integrador en fase
canal_y	8	I	Señal a la salida del integrador en cuadratura
constelacion_x	8	I	Constelación del canal *x* (fase) sin ruido
constelacion_y	8	I	Constelación del canal *y* (cuadratura) sin ruido
constelacion_ruido_x	8	I	Señal que contiene la constelación resultante al integrar la señal con ruido del canal *x*
constelacion_ruido_y	8	I	Señal que contiene la constelación resultante al integrar la señal con ruido del canal *y*
transmite	1	O	Señal del sistema que ordena al módulo la transmisión del dato que se encuentra en **dato_tx_in**. La orden dura un único ciclo de reloj
datos_8bits	8	O	Dato pseudoaleatorio de 8 bits para la generación del ruido

Tabla 12. Puertos de entrada y salida del módulo del control de envío

Como se observa en la Tabla 12, algunas de los puertos de salida que se han de enviar presentan 12 bits, pero en realidad, únicamente se pueden enviar 8 bits por cada paquete, por lo que se han convertido los 8 MSB en complemento a 2 con el comando "*conv_signed"* con el fin de ajustar el formato de los paquetes.

Este módulo presenta un contador que se controla con la señal periódica que indicaba el tiempo en enviarse una trama, de manera que se enviará cada paquete en función del valor que vaya tomando el contador.

Un aspecto a tener en cuenta es que el programa en *LabView* ha sido configurado de manera que el primer dato recibido debe ser "FF" como indicativo de inicio de los paquetes. Por tanto, la estructura de los paquetes es la que se muestra en la Figura 15.

FF	Señal modulada	Dato	Señal demodulada	Canal X	Canal Y	Señal con ruido	Amplitud de ruido	Constelación X	Constelación Y	Constelación X (ruido)	Constelación Y (ruido)

Figura 15. Estructura de los paquetes.

Además, este módulo también presenta un detector de flanco para la señal **transmite** con el fin de que esta señal dure un ciclo de reloj para el buen funcionamiento del módulo de transmisión de la UART. El envío de paquetes se realizará cuando la señal **transmitiendo** se encuentre a '0' y la señal **transmite** esté a '1', marcado por el tiempo de envío de cada trama.

3.2. Diseño en LabView

En el anexo 8.2 se muestra tanto el panel principal del software del entrenador como su diagrama de bloques. En los siguientes apartados, se explicarán cada uno de los elementos que componen el sistema, los cuales se encuentran dentro de un bucle *while* infinito con el fin de que la UART se ejecute sin interrupciones.

3.2.1. Bloque de transmisión y recepción de datos

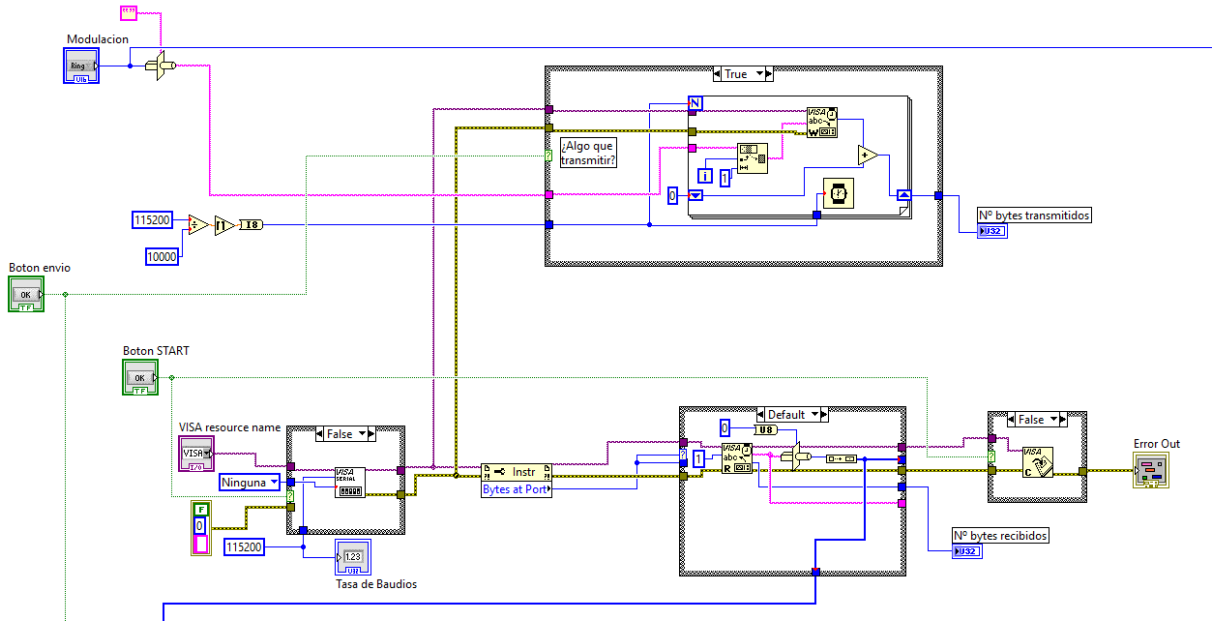


Figura 16. Configuración de la transmisión y recepción de datos de la UART en LabView

La transmisión y recepción de datos a través de la UART es posible gracias a una *Application Programming Interface (*API*)* que proporciona una interfaz de programación para controlar instrumentos USB, Ethernet, seriales… en entornos de desarrollo de aplicaciones de *National Instruments*, como es el caso de *LabView* [7].

Lo que se ha hecho en la Figura 16 ha sido inicializar el puerto serial especificando el tipo de paridad (ninguna), la tasa de baudios (115 200 baudios) y el puerto serie al que va conectado, el cual es seleccionado por el usuario en el panel frontal. También es posible introducir el número de bits de datos a enviar o recibir, pero como por defecto este valor es 8, no se ha especificado. Además, la inicialización se lleva a cabo una vez que se presione el botón de START.

Luego se tiene por un lado el bloque encargado de transmitir, que consta de la función "*VISA Write Function*" la cual escribe los datos desde un *buffer* de escritura al dispositivo especificado por el nombre del recurso VISA (puerto seleccionado). Lo único que se va a enviar por el puerto serial desde el software es el tipo de modulación seleccionada, los cuales están definidos cada uno por un número en binario de 8 bits. Estos bits deben ser enviados uno a uno, por tanto, se ha realizado la funcionalidad de ir mandando los bits de la cadena de *string* del dato a enviar mediante la función "*String Subset Function*" y un bucle *for*.

Por otro lado, el bloque encargado de recibir consta de la función "*VISA Read Function*" que se encarga de leer el número especificado de bytes desde el puerto serial, en este caso se ha especificado que los vaya leyendo byte a byte, y devuelve los datos leídos al *buffer* de escritura.

Por último, este bloque también presenta la funcionalidad de cerrar la sesión con el dispositivo mediante la función "*VISA Close Function*" cuando el botón de START está inactivo o cuando haya algún error en la comunicación.

3.2.2. Extracción de los datos de los paquetes

De los paquetes que van llegando por la API de VISA, hay que diferenciar cuándo empieza y termina la trama de paquetes. Para ello, se ha utilizado la subrutina de la Figura 17, que consiste en ir buscando un carácter de control, en este caso "FF", e ir agrupando en columnas cada paquete. Una vez que se vuelve a encontrar el carácter de control, en la siguiente columna se vuelven a organizar los paquetes, de manera que se crea una matriz cuyas columnas son el tipo de paquete recibido y las filas las muestras temporales adquiridas por cada una de las señales.

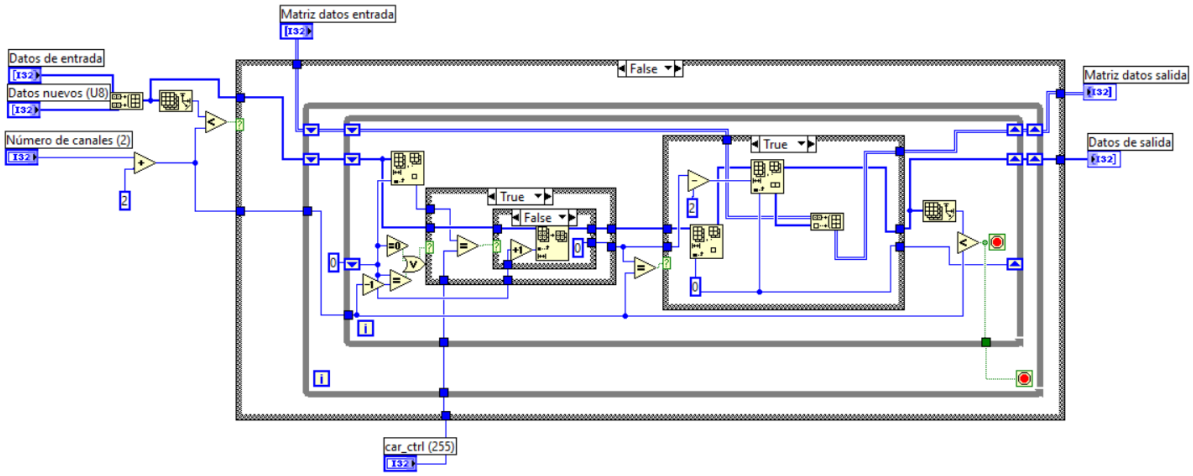


Figura 17. Subrutina de la búsqueda del carácter de control "FF"

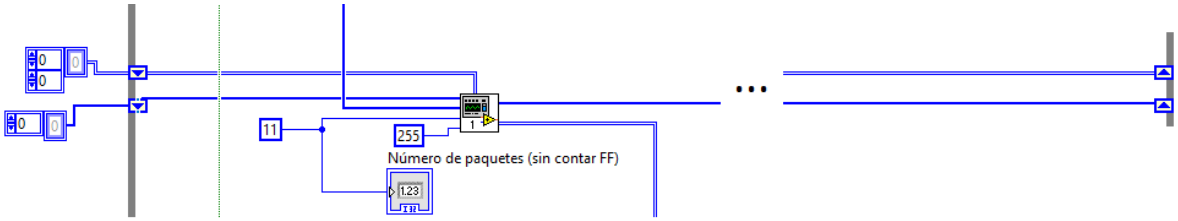


Figura 18. Utilización de la Subrutina y de los registros de desplazamiento en LabView

Ahora bien, una vez colocada la subrutina en el diagrama de bloques del proyecto (Figura 18), se ha utilizado registros de desplazamiento con el fin de acceder a datos desde iteraciones anteriores del bucle *while* infinito. Estos registros aparecen como un par de terminales directamente opuestas en las partes verticales del borde del bucle. El terminal en la parte derecha del bucle contiene una flecha hacia arriba y almacena datos de la terminación de una iteración mientras que la flecha que apunta hacia debajo de la parte izquierda contiene los datos de la anterior iteración [8]. Con estos registros se va construyendo la matriz con las muestras de cada paquete organizados de la manera que se ha comentado anteriormente.

3.2.3. Muestras a visualizar y limpieza de memoria

Antes de visualizar las muestras, debe haber una limpieza del *array* que contiene las muestras a partir de un cierto valor de muestras totales o incluso mediante una orden ejecutada por el usuario. Por ello, en el primer bloque *if* de la Figura 19, se resetea el *array* mediante la pulsación de los pulsadores de START y de "Limpiar gráficas", con el uso de la función "*Delete From Array*".

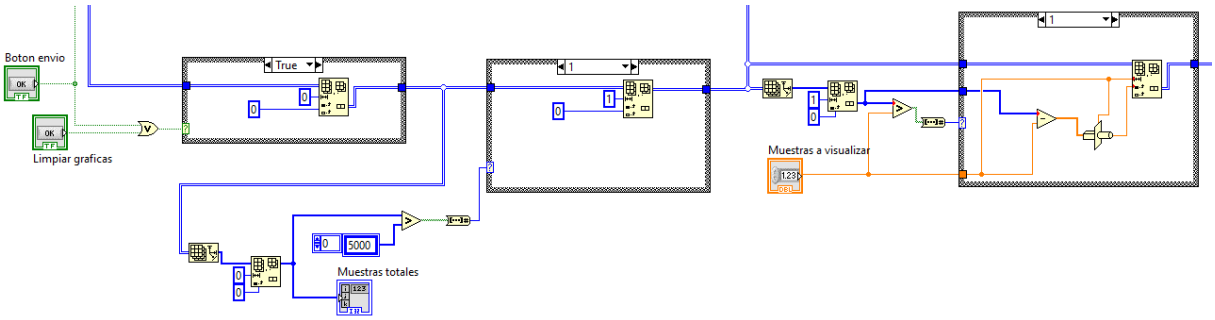


Figura 19. Limpieza de arrays según el caso en el programa LabView

La siguiente sentencia *if* utiliza la misma funcionalidad, pero en este caso va borrando las primeras muestras cuando llega a 5000 muestras totales. Es decir, cuando llega a esta cifra, se irán borrando las primeras muestras que se han obtenido mientras que las nuevas se continuarán añadiendo, manteniéndose por tanto una memoria únicamente de las últimas 5000 muestras recibidas.

Por último, la siguiente sentencia *if* devuelve la matriz con el número de muestras que ha seleccionado el usuario en el panel frontal con el fin de visualizar esa cantidad de datos en las gráficas.

3.2.4. Escalado y visualización de las muestras

Como las muestras toman valores de 0 a 255, se ha escalado con el fin de darle un valor lógico y mostrarlo a través de las gráficas. Además, se ha introducido la posibilidad de controlar la ganancia y el desplazamiento de las señales a visualizar en las gráficas como se puede observar en la Figura 20 y en la Figura 21.

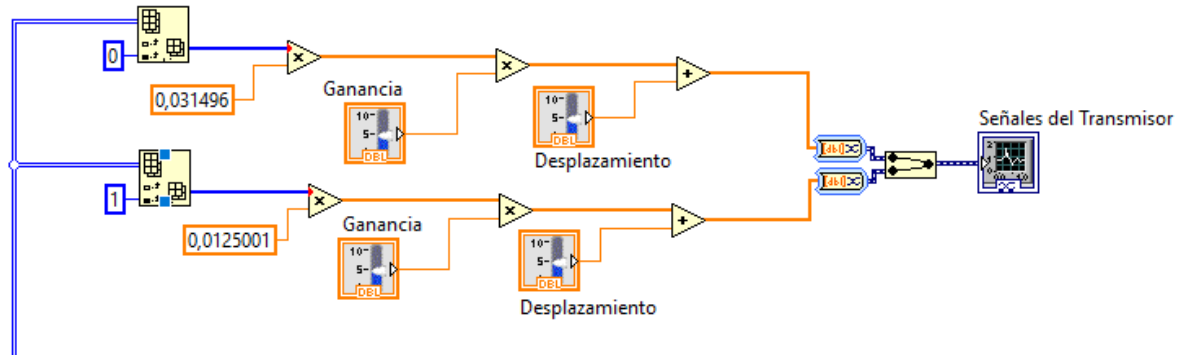


Figura 20. Escalado y visualización de las muestras del transmisor en LabView.

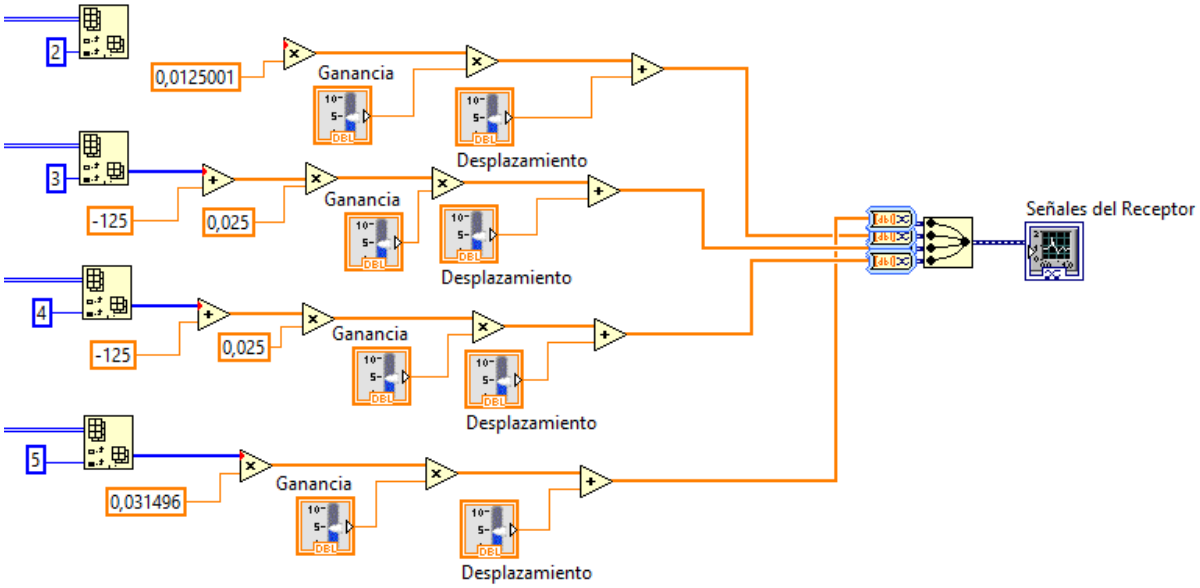


Figura 21. Escalado y visualización de las muestras del receptor en LabView.

En la Figura 22 se muestra cómo se han configurado los valores de la distribución de ruido. Se ha utilizado la función “Histogram VI” para generar un histograma que permita la visualización de las muestras de ruido formando la forma de la distribución de probabilidad gaussiana.

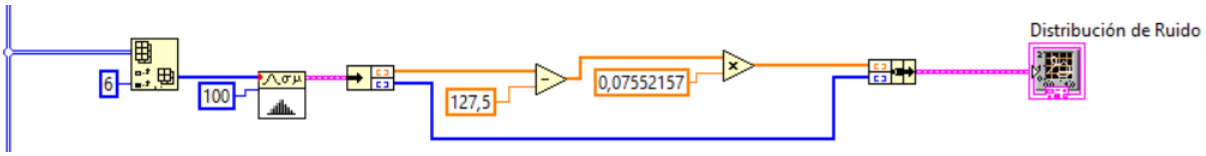


Figura 22. Configuración de los valores de la distribución de ruido en LabView

Para el caso de las constelaciones, hay dos sentencias *case* (Figura *23*) ya que, para cada modulación, los valores de las constelaciones van cambiando. Por lo cual se ha escogido este tipo de estructura que depende del valor de la modulación, y así escalar con unos valores u otros para la correcta visualización en las gráficas. Cabe destacar que, para la representación de las constelaciones, se ha utilizado el tipo de gráfica XY.

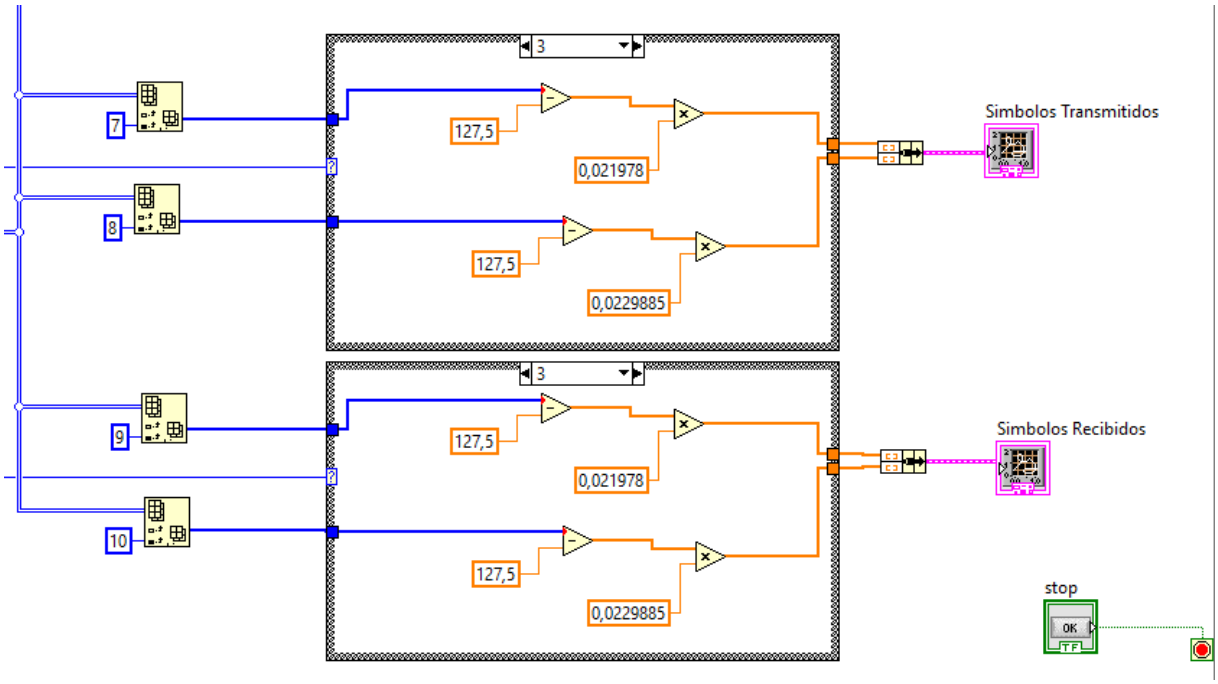


Figura 23. Escalado y visualización de las constelaciones en función del tipo de modulación en LabView

Capítulo 4. Análisis de resultados con el entrenador implementado

Capítulo 4. Análisis de resultados con el entrenador implementado

En este capítulo se profundizará sobre los códigos realizados, la adaptación del nuevo entrenador y las simulaciones para observar el comportamiento de las señales. Además, se analizarán los recursos utilizados por ambos entrenadores y se mostrarán los resultados en el software de LabView.

4.1. Análisis en Vivado

4.1.1. Módulo de transmisión UART

Para diseñar el divisor de frecuencia del transmisor de la UART (Figura 24) se dividió las frecuencias de entrada y de salida, que se corresponden a la frecuencia de la placa y a la tasa de baudios elegida (115 200 baudios), respectivamente.

$$Cuenta = \frac{100 \cdot 10^6 \text{Hz}}{115200 \text{ bits/s}} = 868{,}0555 \text{ cuentas} \approx 868 \text{ cuentas}$$

Al redondear, se obtiene una frecuencia a la salida de 115 207,37 Hz. Que este valor no sea exacto no presenta ningún inconveniente para el diseño.

$$Frecuencia\ de\ salida = \frac{100 \cdot 10^6 \text{Hz}}{868 \text{ cuentas}} = 115207{,}37 \text{ Hz} \rightarrow Periodo = 8{,}68\ \mu s$$

Este valor corresponde a la cuenta necesaria para obtener la frecuencia de salida, la cual se almacena en la constante **c_fin_cont_baud**. Además, se ha creado una función en la librería "*UART_PKG"* que calcula cuántos bits se necesitan para la señal del contador (**cont_baud**), calculando el logaritmo en base dos del valor del fin de cuenta.

Esta función se muestra en la Figura 25, la cual consiste en dividir por dos de manera sucesiva hasta que el resultado sea cero. El número de divisiones que se hayan realizado indicará el número de bits necesarios. Esta función devuelve un número menor al número de bits necesario, por lo que, al representar un entero, se tendría que sumar una unidad al resultado, sin embargo, para los tipos *unsigned* no haría falta porque el rango incluye el cero.

```
P_DivFreq: Process(reset,clk)
begin
    if reset = '1' then -- Se ponen a cero las señales cuando el reset esté activo.
        cont_baud <= (others => '0');
        baud <= '0';
    elsif clk'event and clk='1' then
        if en_divfreq = '1' then    -- Solo hace la cuenta cuando la señal enable esté activa
            baud <= '0';
            if cont_baud = c_fin_cont_baud then -- Cuando llega al final de la cuenta, se reinicia,
                                                -- y 'baud' se pone a 1 (que indica el tiempo de envio de un bit)
                baud <= '1';
                cont_baud <= (others => '0');
            else
                cont_baud <= cont_baud + 1;
            end if;
        else
            cont_baud <= (others => '0');
        end if;
    end if;
end process;
```

Figura 24. Fragmento del código del divisor de frecuencia del módulo de transmisión de la UART

```
Package body UART_PKG is
    function log2i (valor : positive) return natural is
        variable tmp, log2: natural;
    begin
        tmp := valor / 2; -- division entera, redondea al entero inmediatamente menor o =
        log2 := 0;
        while (tmp /= 0) loop
            tmp := tmp/2;
            log2 := log2 + 1;
        end loop;
        return log2;
    end function log2i;
end UART_PKG;
```

Figura 25. Fragmento del código que calcula el rango de la señal del contador del divisor de frecuencia

Por otro lado, en la Figura 26 se muestra el proceso combinacional que obtiene el estado siguiente a partir del estado actual y las entradas, y el proceso secuencial que actualiza el estado actual en cada ciclo de reloj, guardándolo en un biestable tipo D. Esta máquina de estados es de tipo Mealy, es decir, las salidas dependen del estado actual y de las entradas.

En la Figura 27 se muestra el fragmento del código de la máquina de estados que controla el valor de las salidas.

```
P_COMB_ESTADO : process(estado_actual, transmite, baud, fin_cont8bits)
begin
    case estado_actual is
        when e_init =>
            if transmite = '1' then
                estado_siguiente <= e_bit_init;
            else
                estado_siguiente <= e_init;
            end if;
        when e_bit_init =>
            if baud = '1' then
                estado_siguiente <= e_bits_dato;
            else
                estado_siguiente <= e_bit_init;
            end if;
        when e_bits_dato =>
            if fin_cont8bits = '1' then
                estado_siguiente <= e_bit_fin;
            else
                estado_siguiente <= e_bits_dato;
            end if;
        when e_bit_fin =>
            if baud = '1' then
                estado_siguiente <= e_init;
            else
                estado_siguiente <= e_bit_fin;
            end if;
    end case;
end process;
```

```
P_SEQ_FSM : process(clk, reset)
begin
    if reset = '1' then
        estado_actual <= e_init;
    elsif clk'event and clk = '1' then
        estado_actual <= estado_siguiente;
    end if;
end process;
```

Figura 26. Fragmentos del código que controlan las transiciones de la máquina de estados del transmisor de la UART

```
P_COM_SALIDAS : process(estado_actual, transmite, fin_cont8bits, baud)
begin
    case estado_actual is
        when e_init =>
            estado_tx <= "00";
            desplaza <= '0';
            en_divfreq <= '0';
            transmitiendo <= '0';
            if transmite = '1' then
                cargadato <= '1';
            else
                cargadato <= '0';
            end if;
        when e_bit_init =>
            estado_tx <= "01";
            cargadato <= '0';
            desplaza <= '0';
            en_divfreq <= '1';
            transmitiendo <= '1';
        when e_bits_dato =>
            estado_tx <= "10";
            cargadato <= '0';
            en_divfreq <= '1';
            transmitiendo <= '1';
            if fin_cont8bits = '0' then
                desplaza <= baud;
            else
                desplaza <= '0';
            end if;
        when e_bit_fin =>
            estado_tx <= "11";
            cargadato <= '0';
            desplaza <= '0';
            en_divfreq <= '1';
            transmitiendo <= '1';
    end case;
end process;
```

Figura 27. Fragmento del código donde controlan las salidas de la máquina de estados del transmisor de la UART.

El contador de los bits de datos que se muestra en la Figura 28, únicamente cuenta cuando se encuentra en el estado **e_bits_dato** (asignado como “10”) y cuando pasa el tiempo correspondiente al envío de un bit, es decir, cuando la señal **baud** se encuentra activa.

```
begin

process(clk, reset, baud, estado_tx, cont)
begin

    if reset = '1' then
        cont <= (others => '0');
    elsif clk'event and clk='1' then
        if estado_tx = "10" and baud = '1' then
            if cont = 7 then
                cont <= (others => '0');
            else
                cont <= cont + 1;
            end if;
        end if;
    end if;
end process;

fin_cont8bits <= '1' when (cont = 7 and baud ='1') else '0';

end Behavioral;
```

Figura 28. Fragmento de código del contador de bits de los datos a enviar

Luego, el registro de desplazamiento de carga paralela (Figura 29) consiste en desplazar a la izquierda los bits guardados en la señal **dato_tx_reg** cuando la señal **desplaza** se encuentre activa a alta. Los bits de datos se almacenarán en dicha señal cuando la señal **cargadato** se encuentre activa. Al desplazar a la izquierda, lo que haya en el bit 7 se perderá, y lo que haya en el bit cero se mantendrá, es decir, no se le asignará ningún valor nuevo [3]. El fragmento del código que se corresponde al registro de desplazamiento se observa en la Figura 30.

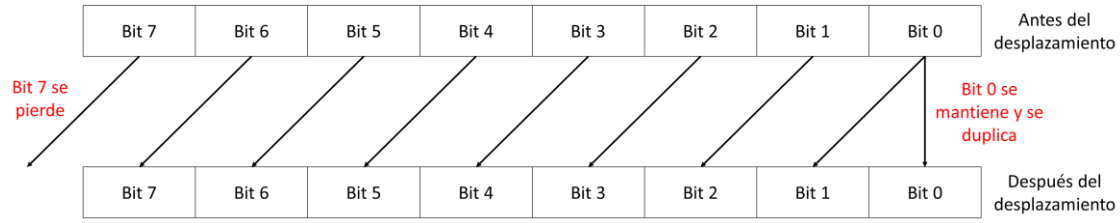


Figura 29. Registro de desplazamiento de carga paralela hacia la izquierda

```
P_registro : Process (clk, reset)
begin
    if reset = '1' then
        dato_tx_reg <= (others => '0');
    elsif clk'event and clk = '1' then
        if cargadato = '1' then
            dato_tx_reg <= dato_tx_in;
        elsif desplaza = '1' then
            dato_tx_reg(6) <= dato_tx_reg(7);
            dato_tx_reg(5) <= dato_tx_reg(6);
            dato_tx_reg(4) <= dato_tx_reg(5);
            dato_tx_reg(3) <= dato_tx_reg(4);
            dato_tx_reg(2) <= dato_tx_reg(3);
            dato_tx_reg(1) <= dato_tx_reg(2);
            dato_tx_reg(0) <= dato_tx_reg(1);
        end if;
        dato_tx <= dato_tx_reg(0);
    end if;
end process;
```

Figura 30. Fragmento de código del registro de desplazamiento PIPO

Por último, en la Figura 31 se muestra el módulo de selección que elige el bit que se debe enviar en cada momento según el estado en el que se encuentre.

```
process (clk, reset, estado_tx, dato_tx)
begin
    if reset = '1' then
        fpga_tx <= '1';
    elsif clk'event and clk = '1' then
        case estado_tx is
            when "00" => fpga_tx <= '1';        -- Estado reposo
            when "01" => fpga_tx <= '0';        -- Estado bit de inicio
            when "10" => fpga_tx <= dato_tx;    -- Estado bits de datos
            when others => fpga_tx <= '1';      -- Estado bit final
        end case;
    end if;

end process;
```

Figura 31. Fragmento del código del módulo de selección.

Como se puede observar en la Figura 32, en la señal **dato_tx_in** se aprecia el dato de 8 bits a transmitir (“10000110”). Cuando la señal **transmite** se pone en alta un ciclo de reloj, se inicia el proceso de serialización. Puede comprobarse el envío del valor señalado en la línea **fpga_tx** del cronograma.

Figura 32. Simulación de la transmisión mediante UART del dato “10000110”

4.1.2. Módulo de recepción de la UART

En la Figura 33 se muestra el doble registro de la señal **fpga_rx**, que correspondería a dos registros tipo D en cascada.

```
P_DETECTA: process (reset, clk)
begin
    if reset = '1' then
        aux <= '1';
        fpga_rx_rg <= '1';
    elsif clk'event and clk = '1' then
        aux <= fpga_rx;
        fpga_rx_rg <= aux;
    end if;
end process;
```

Figura 33. Fragmento del código del registro de entrada del módulo de recepción de la UART.

El divisor de frecuencia del módulo de recepción (Figura 34) es similar al del transmisor, diferenciándose en que esta vez tiene dos salidas: **baud** y **baud_medio**. Esta última marca el punto medio de cada bit de la transmisión, por tanto, la cuenta final sería la mitad que para obtener la señal **baud**, es decir, 434 cuentas.

```
P_DivFreq: Process(reset,clk)
begin
    if reset = '1' then
        cont_baud <= (others => '0');
        baud <= '0';
        baud_medio <= '0';
    elsif clk'event and clk='1' then
        if en_divfreq = '1' then
            baud <= '0';
            baud_medio <= '0';
            if cont_baud = c_fin_cont_baud then
                baud <= '1';
                cont_baud <= (others => '0');
            elsif cont_baud = c_fin_cont_baud_medio then
                baud_medio <= '1';
                cont_baud <= cont_baud + 1;
            else
                cont_baud <= cont_baud + 1;
            end if;
        else
            cont_baud <= (others => '0');
        end if;
    end if;
end process;
```

Figura 34. Fragmento del código del divisor de frecuencia del módulo de recepción de la UART

Por otro lado, en la Figura 35 y Figura 36 se muestran los fragmentos del código correspondientes a la configuración de la máquina de estado para el módulo de control de recepción de la UART.

```
P_COMB_ESTADO : process(estado_actual, baud, fpga_rx_rg, fin_cont8bits)
begin
    case estado_actual is
        when e_init =>
            if fpga_rx_rg = '0' then
                estado_siguiente <= e_bit_init;
            else
                estado_siguiente <= e_init;
            end if;
        when e_bit_init =>
            if baud = '1' then
                estado_siguiente <= e_bits_dato;
            else
                estado_siguiente <= e_bit_init;
            end if;
        when e_bits_dato =>
            if fin_cont8bits = '1' then
                estado_siguiente <= e_bit_fin;
            else
                estado_siguiente <= e_bits_dato;
            end if;
        when e_bit_fin =>
            if baud = '1' then
                estado_siguiente <= e_init;
            else
                estado_siguiente <= e_bit_fin;
            end if;
    end case;
end process;
```

```
P_SEQ_FSM : process(clk, reset)
begin
    if reset = '1' then
        estado_actual <= e_init;
    elsif clk'event and clk = '1' then
        estado_actual <= estado_siguiente;
    end if;
end process;
```

Figura 35. Fragmentos del código que controlan las transiciones de la máquina de estados del receptor de la UART

En la Figura 37 se muestra el registro de desplazamiento SIPO, opuesto al del transmisor, el cual va cargando los datos en serie y los devuelve en paralelo. El desplazamiento se realiza cuando la señal **desplaza** se encuentra activo en alta.

Este módulo como ya se mencionó, presenta también un contador de los bits de datos que es el mismo que se mostró para el transmisor.

```
P_COM_SALIDAS : process(estado_actual, fin_cont8bits, baud, baud_medio)
begin
    case estado_actual is
        when e_init =>
            estado_tx <= "00";
            en_divfreq <= '0';
            recibiendo <= '0';
            aviso_rx <= '0';
            desplaza <= '0';
        when e_bit_init =>
            estado_tx <= "01";
            en_divfreq <= '1';
            recibiendo <= '1';
            aviso_rx <= '0';
            desplaza <= '0';
            if baud_medio = '1' then
                desplaza <= baud_medio;
            else
                desplaza <= '0';
            end if;
        when e_bits_dato =>
            estado_tx <= "10";
            en_divfreq <= '1';
            recibiendo <= '1';
            aviso_rx <= '0';
            if fin_cont8bits = '0' then
                desplaza <= baud_medio;
            else
                desplaza <= '0';
            end if;
        when e_bit_fin =>
            estado_tx <= "11";
            en_divfreq <= '1';
            desplaza <= '0';
            if baud = '1' then
                aviso_rx <= '1';
                recibiendo <= '0';
            else
                aviso_rx <= '0';
                recibiendo <= '1';
            end if;
    end case;
end process;
```

Figura 36. Fragmento del código donde controlan las salidas de la máquina de estados del receptor de la UART.

```
P_registro : Process (clk, reset)
begin
    if reset = '1' then
        dato_rx_reg <= (others => '0');
    elsif clk'event and clk = '1' then
        if desplaza = '1' then
            dato_rx_reg(7) <= fpga_rx_rg;
            dato_rx_reg(6) <= dato_rx_reg(7);
            dato_rx_reg(5) <= dato_rx_reg(6);
            dato_rx_reg(4) <= dato_rx_reg(5);
            dato_rx_reg(3) <= dato_rx_reg(4);
            dato_rx_reg(2) <= dato_rx_reg(3);
            dato_rx_reg(1) <= dato_rx_reg(2);
            dato_rx_reg(0) <= dato_rx_reg(1);
        end if;
    end if;
end process;

dato_rx_out <= dato_rx_reg;
```

Figura 37. Fragmento de código del registro de desplazamiento SIPO

Como se puede observar en la Figura 38, cuando el dato se ha cargado completamente, la señal **aviso_rx** se activa confirmando que el dato está disponible en la señal **dato_rx_out**, lo que hace que estos datos se almacenen en la señal **modulación** en el módulo *Top* del entrenador.

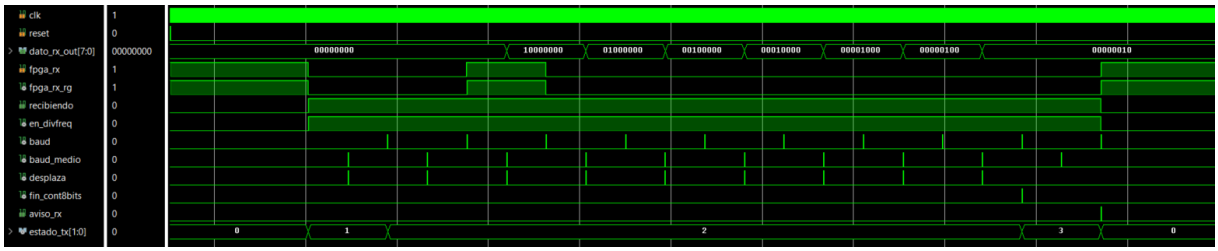


Figura 38. Simulación de la recepción mediante UART del dato "00000010"

En la Figura 39 se muestra el fragmento del código que se encuentra en el módulo *Top* el cual asigna el valor de la modulación a partir del dato recibido mediante la UART.

```
process(clk, reset)
begin

    if reset = '1' then
        modulacion <= (others => '0');
    elsif clk'event and clk='1' then
        if aviso_rx = '1' then
            modulacion <= dato_rx_out;
        end if;
    end if;

end process;
```

Figura 39. Fragmento del código de la obtención del valor de la modulación a partir del módulo de recepción de la UART (módulo Top)

4.1.3. Módulo del generador de datos

Para diseñar el generador de datos pseudoaleatorio, se ha utilizado un bucle *for* de manera que, para el primer caso, la entrada del biestable sea la salida de la puerta XOR de las señales correspondiente a retroalimentar y el resto se conecten las salidas con sus entradas. Con esto se consigue evitar que el código sea muy extenso.

En la Figura 40 se muestran los generadores de datos tanto para las modulaciones como para el ruido, con la diferencia de que para el primero la puerta XOR se realiza con los biestables 2 y 5 mientras que para el segundo se realiza con los biestables 19 y 21. Además, las señales de reloj son distintas. Esto es porque la

señal modulada se genera mediante los punteros cuyos desplazamientos vienen marcadas por la señal de reloj de la placa, por tanto, el ruido se debe añadir en cada desplazamiento de estos punteros.

```
Generador_datos : for I in 0 to Nreg - 1 generate
    Reg00 : if (I=0) generate
        Reg0 : Registro
        port map (
            clk => clk_periodo,
            preset => reset,
            D => sig_xor,
            Q => Q_int(0)
        );
    end generate;
    Regs : if I>0 generate
        Reg : Registro
        port map (
            clk => clk_periodo,
            preset => reset,
            D => Q_int(I-1),
            Q => Q_int(I)
        );
    end generate;
end generate;

dato <= Q_int(Nreg-1);
dato_2bits <= Q_int(1 downto 0);
sig_xor <= Q_int(1) xor Q_int(Nreg-1);
```

```
Generador_datos_8bits : for I in 0 to Nreg_8 - 1 generate
    Reg00 : if (I=0) generate
        Reg0 : Registro
        port map (
            clk => clk,
            preset => reset,
            D => sig_xor_8,
            Q => Q_int_8(0)
        );
    end generate;
    Regs : if I>0 generate
        Reg : Registro
        port map (
            clk => clk,
            preset => reset,
            D => Q_int_8(I-1),
            Q => Q_int_8(I)
        );
    end generate;
end generate;

dato_8bits <= Q_int_8(7 downto 0);
sig_xor_8 <= Q_int_8(20) xor Q_int_8(18);
```

Figura 40. Fragmento del código que genera los datos pseudoaleatorios, siendo el código de la izquierda para las modulaciones y el de la derecha para el ruido

En el bucle *for* se han instanciado biestables tipo D (Figura 41), que son inicializados a ‘1’ debido a que se provocaría una salida a nivel lógico bajo de manera indefinida si se inicializaran a ‘0’, como ya se comentó anteriormente.

```
process(clk, preset)
begin
    if preset = '1' then
        Q <= '1';
    elsif clk'event and clk = '1' then
        Q <= D;
    end if;
end process;
```

Figura 41. Fragmento del código del registro que se ha instanciado

En la Figura 42 se muestran los datos pseudoaleatorios generados para realizar las modulaciones. Obsérvese que los datos van cambiando por cada ciclo de la señal **clk_periodo**. Luego, en la Figura 43, se muestran los datos pseudoaleatorios (en hexadecimal) para el ruido que van cambiando su valor en función de los ciclos de la señal de reloj de la placa.

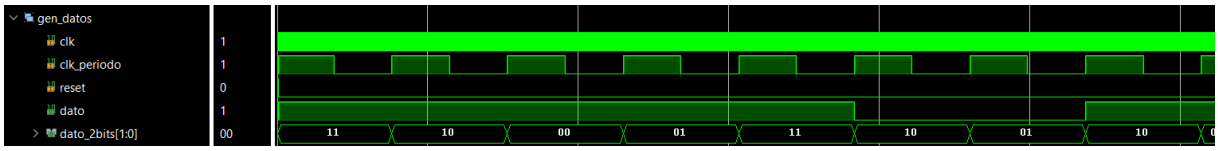


Figura 42. Simulación de las señales del generador de datos para la realización de las modulaciones

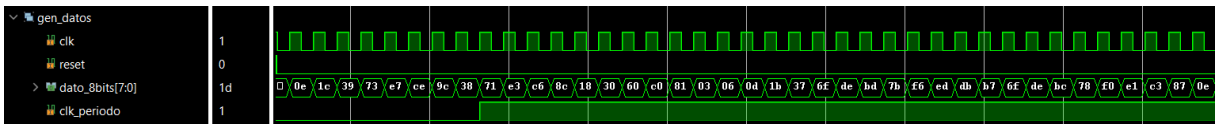


Figura 43. Simulación de las señales del generador de datos para la generación de ruido

4.1.4. Módulo del modulador

La señal **clk_periodo** que controla el cambio de valor de los datos pseudoaleatorios que deben transmitirse, la crea el módulo del modulador (Figura 44). Haciendo que la cuenta llegue a 64 se consigue que durante un bit de dato se transmitan dos ciclos de la señal portadora, ya que la tabla que define la portadora consta de M = 32 posiciones, es decir, 32 valores que representan un ciclo completo de la portadora senoidal a intervalos de $2\pi/M$ radianes desde 0 hasta $2\pi(M-1)/M$.

```
process(clk, reset, cont_periodo)
begin
   if(reset = '1') then
       cont_periodo <= 0;
       aux_periodo <= '1';
   elsif clk'event and clk = '1' then
       if cont_periodo = 63 then
           cont_periodo <= 0;
           aux_periodo <= not(aux_periodo);
       elsif cont_periodo = 31 then
           aux_periodo <= not(aux_periodo);
           cont_periodo <= cont_periodo + 1;
       else
           cont_periodo <= cont_periodo + 1;
       end if;
   end if;
end process;
```

Figura 44. Fragmento del código que genera una señal periódica que marca los ciclos que tiene la señal en un bit de dato

En la Figura 45 se muestra la creación de los punteros para obtener las diferentes señales portadoras. En este caso, se necesitan la señal seno para obtener la señal modulada en ASK y BPSK, la señal seno de doble frecuencia para obtener la señal modulada en FSK y las señales desfasadas ciertos ángulos para la modulación QPSK.

```
process(clk, reset, modulacion, dato, dato_2bits)
begin
   if(reset = '1') then
       puntero <= 0;
       puntero_espacio <= 0;
       constelacion_x <= (others => '0');
       constelacion_y <= (others => '0');
       valor <= (others => '0');

   elsif clk'event and clk = '1' then

       puntero <= (puntero + 1) mod M;
       puntero_espacio <= (puntero_espacio + 2) mod M;

       puntero_45 <= (puntero + 5) mod M;
       puntero_135 <= (puntero + 13) mod M;
       puntero_255 <= (puntero + 21) mod M;
       puntero_315 <= (puntero + 29) mod M;
```

Figura 45. Fragmento del código del reseteo de las señales y la generación de las portadoras

En el mismo proceso, se encuentra el código de la Figura 46, que según el dato que se haya obtenido y la modulación elegida, la señal modulada irá guardando el valor de la portadora que corresponda. Además, también se guardará el valor de las constelaciones, obtenidas del valor final de salida del integrador al acabar el período de integración que coincide con el tiempo de símbolo.

```
if modulacion = "00000000" then -- MODULACION ASK.
    if dato = '1' then
        valor <= tabla_onda(puntero);
        constelacion_x <= "10111111"; -- 191
        constelacion_y <= "10000000"; -- 128
    else
        valor <= (others => '0'); --128
        constelacion_x <= "10000000"; -- 128
        constelacion_y <= "10000000"; -- 128
    end if;
elsif modulacion = "00000001" then -- MODULACION BPSK
    if dato = '1' then
        valor <= tabla_onda(puntero);
        constelacion_x <= "10111111"; -- 191
        constelacion_y <= "10000000"; -- 128
    else
        valor <= -tabla_onda(puntero);
        constelacion_x <= "01000000"; -- 64
        constelacion_y <= "10000000"; -- 128
    end if;
elsif modulacion = "00000010" then -- MODULACION FSK
    if dato = '1' then
        valor <= tabla_onda(puntero);
        constelacion_x <= "10111111"; -- 191
        constelacion_y <= "01111111"; -- 127
    else
        valor <= tabla_onda(puntero_espacio);
        constelacion_x <= "01111111"; -- 127
        constelacion_y <= "10111111"; -- 191
    end if;
```

```
    elsif modulacion = "00000011" then -- QPSK
    -- PARA ESTA MODULACION SE HA UTILIZADO LA CODIFICACION GRAY
        if dato_2bits = "00" then
            valor <= tabla_onda(puntero_45);
            constelacion_x <= "10101101"; -- 173
            constelacion_y <= "10101011"; -- 171
        elsif dato_2bits = "01" then
            valor <= tabla_onda(puntero_135);
            constelacion_x <= "01010010"; -- 82
            constelacion_y <= "10101011"; -- 171
        elsif dato_2bits = "11" then
            valor <= tabla_onda(puntero_255);
            constelacion_x <= "01010010"; -- 82
            constelacion_y <= "01010100"; -- 84
        elsif dato_2bits = "10" then
            valor <= tabla_onda(puntero_315);
            constelacion_x <= "10101101"; -- 173
            constelacion_y <= "01010100"; -- 84
        else
            valor <= (others => '0');
            constelacion_x <= (others => '0');
            constelacion_y <= (others => '0');
        end if;
    else
        valor <= (others => '0');
        constelacion_x <= (others => '0');
        constelacion_y <= (others => '0');
    end if;
  end if;
end process;
```

Figura 46. Fragmento del código de cómo se genera la señal modulada y las constelaciones de las modulaciones en función del dato recibido y el tipo de modulación

En la Figura 47, Figura 48, Figura 49 y Figura 50, se muestran las simulaciones de las distintas modulaciones que pueden analizarse con el entrenador implementado.

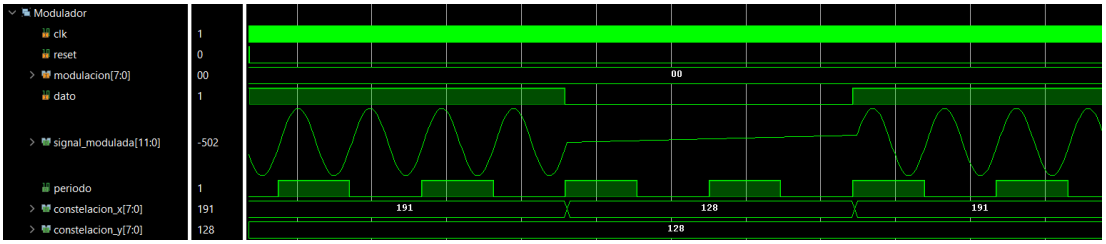


Figura 47. Simulación para la obtención de la señal modulada en ASK

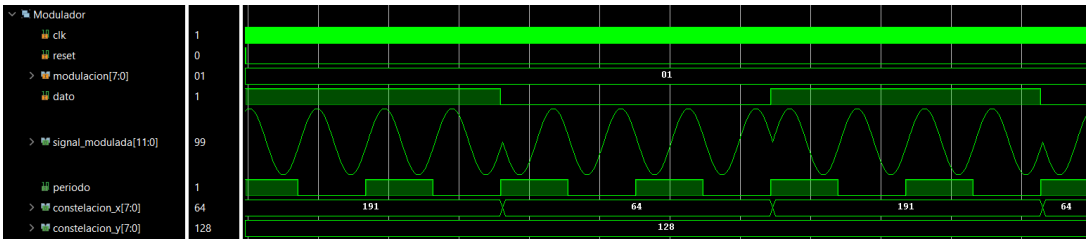


Figura 48. Simulación para la obtención de la señal modulada en BPSK

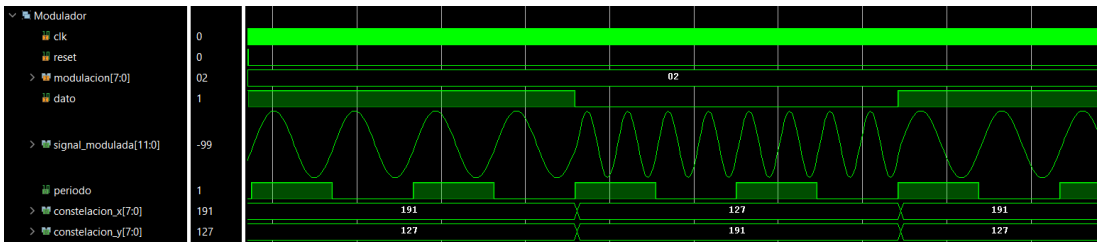


Figura 49. Simulación para la obtención de la señal modulada en FSK

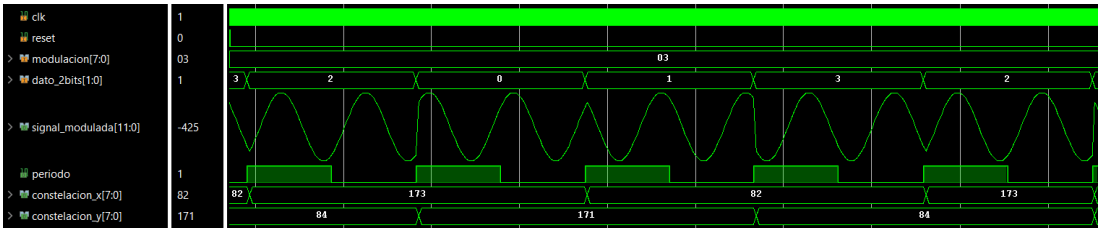


Figura 50. Simulación para la obtención de la señal modulada en QPSK

4.1.5. Módulo del generador de ruido

Para la obtención del ruido se ha creado una tabla con 256 valores que se corresponden a los valores de la distribución de probabilidad gaussiana discretizada. Para discretizar esta señal en 8 bits, los valores de la distribución normal se han

escalado, desplazado y redondeado para convertirlos a valores enteros de 8 bits a partir de la ecuación (3).

$$f[n] = \left(\frac{f(nT_s) - V_{min}}{V_{max} - V_{min} + \varepsilon}\right) \cdot 2^q, \qquad (3)$$

Donde $f(nT_s)$ son los valores de la distribución de probabilidad gaussiana sin discretizar, V_{min} es el valor mínimo, V_{max} es el valor máximo, q es el número de bits y ε es una constante de Matlab que indica la distancia mínima que puede reconocer el programa entre dos números.

Los valores discretizados se han obtenido en Matlab a partir del código del anexo 8.3 y se han introducido en la librería "*real2bit"*, creando así una tabla con dichos valores.

En la Figura 51 se muestra cómo se obtiene la amplitud del ruido a partir del número aleatorio de 8 bits mientras que en la Figura 52 se muestra la ganancia que se le aplica al ruido en función de la posición de los *switches* de la FPGA. Además, este ruido, aparte de guardarla en una variable para enviar estos datos a través de la UART, se suma a la señal modulada para que se observe el ruido en la propia señal.

```
process(clk, reset)
begin
    if(reset = '1') then
        ruido <= (others => '0');
    elsif clk'event and clk = '1' then
        ruido <= tabla_distribucion(conv_integer(unsigned(dato_8bits)));
    end if;
end process;
```

Figura 51. Fragmento del código de la obtención de la amplitud de ruido a partir de la tabla de distribución normal

```
process(sw_ruido, signal_modulada, ruido)
begin
    if sw_ruido = "0000" then
        signal_ruido <= signal_modulada;
        amplitud_ruido <= (others => '0');
    elsif sw_ruido = "0001" then
        signal_ruido <= signal_modulada + extraer(ruido*truncar(0.066667*1));
        amplitud_ruido <= extraer(ruido*truncar(0.066667*1));
    elsif sw_ruido = "0010" then
        signal_ruido <= signal_modulada + extraer(ruido*truncar(0.066667*2));
        amplitud_ruido <= extraer(ruido*truncar(0.066667*2));
    elsif sw_ruido = "0011" then
        signal_ruido <= signal_modulada + extraer(ruido*truncar(0.066667*3));
        amplitud_ruido <= extraer(ruido*truncar(0.066667*3));
    elsif sw_ruido = "0100" then
        signal_ruido <= signal_modulada + extraer(ruido*truncar(0.066667*4));
        amplitud_ruido <= extraer(ruido*truncar(0.066667*4));
    elsif sw_ruido = "0101" then
        signal_ruido <= signal_modulada + extraer(ruido*truncar(0.066667*5));
        amplitud_ruido <= extraer(ruido*truncar(0.066667*5));
    elsif sw_ruido = "0110" then
        signal_ruido <= signal_modulada + extraer(ruido*truncar(0.066667*6));
        amplitud_ruido <= extraer(ruido*truncar(0.066667*6));
    elsif sw_ruido = "0111" then
        signal_ruido <= signal_modulada + extraer(ruido*truncar(0.066667*7));
        amplitud_ruido <= extraer(ruido*truncar(0.066667*7));
    elsif sw_ruido = "1000" then
        signal_ruido <= signal_modulada + extraer(ruido*truncar(0.066667*8));
        amplitud_ruido <= extraer(ruido*truncar(0.066667*8));
    elsif sw_ruido = "1001" then
        signal_ruido <= signal_modulada + extraer(ruido*truncar(0.066667*9));
        amplitud_ruido <= extraer(ruido*truncar(0.066667*9));
    elsif sw_ruido = "1010" then
        signal_ruido <= signal_modulada + extraer(ruido*truncar(0.066667*10));
        amplitud_ruido <= extraer(ruido*truncar(0.066667*10));
    elsif sw_ruido = "1011" then
        signal_ruido <= signal_modulada + extraer(ruido*truncar(0.066667*11));
        amplitud_ruido <= extraer(ruido*truncar(0.066667*11));
    elsif sw_ruido = "1100" then
        signal_ruido <= signal_modulada + extraer(ruido*truncar(0.066667*12));
        amplitud_ruido <= extraer(ruido*truncar(0.066667*12));
    elsif sw_ruido = "1101" then
        signal_ruido <= signal_modulada + extraer(ruido*truncar(0.066667*13));
        amplitud_ruido <= extraer(ruido*truncar(0.066667*13));
    elsif sw_ruido = "1110" then
        signal_ruido <= signal_modulada + extraer(ruido*truncar(0.066667*14));
        amplitud_ruido <= extraer(ruido*truncar(0.066667*14));
    elsif sw_ruido = "1111" then
        signal_ruido <= signal_modulada + ruido;
        amplitud_ruido <= ruido;
    else
        signal_ruido <= signal_modulada;
        amplitud_ruido <= (others => '0');
    end if;
end process;
```

Figura 52. Fragmento del código de la obtención de la señal de ruido y del control de la ganancia del ruido a partir de los switches de la placa

En la Figura 53 se muestra la comparación entre la señal modulada sin ruido y la misma una vez añadido el mismo.

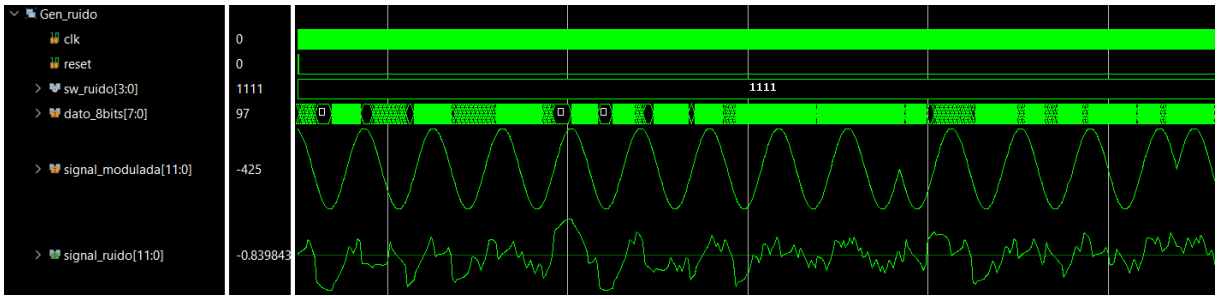


Figura 53. Simulación del módulo de generación de ruido

4.1.6. Módulo del demodulador

En el demodulador, por un lado, se encuentra la generación de portadoras seno y coseno para el caso de las modulaciones ASK, BPSK y QPSK, mientras que para la modulación FSK las portadoras son la onda seno y la onda seno de doble frecuencia (Figura 54).

```
process(clk, reset, modulacion)
begin
    if(reset = '1') then
        portadora_canal_x <= (others => '0');
        portadora_canal_y <= (others => '0');
    elsif clk'event and clk = '1' then
        puntero <= (puntero + 1) mod M;                 -- SENO
        puntero_espacio <= (puntero_espacio + 2) mod M; -- SENO DOBLE FREQ
        puntero_coseno <= (puntero + 9) mod M;          -- COSENO.

        if modulacion = "00000010" then     -- FSK
            portadora_canal_x <= tabla_onda(puntero);
            portadora_canal_y <= tabla_onda(puntero_espacio);
        else                                -- ASK, BPSK (Solo se utiliza el canal x) y QPSK
            portadora_canal_x <= tabla_onda(puntero);
            portadora_canal_y <= tabla_onda(puntero_coseno);
        end if;
    end if;
end process;
```

Figura 54. Fragmento del código para obtener las portadoras de los canales en función del tipo de modulación.

Para resolver el problema de los retardos al realizar las operaciones de multiplicación y la suma acumulativa del integrador, se registró la señal **cont_periodo** tres veces (Figura 55). Esta señal es el valor de cuenta en la que, cuando llega al final del conteo que marca el final del tiempo de integración, resetea la señal del integrador para volver a generar una nueva rampa. Esta señal se corresponde a lo que dura el tiempo de símbolo.

```
process(reset, clk, cont_periodo)
begin
    if(reset = '1') then
        cont_periodo <= 0;
    elsif clk'event and clk = '1' then
        if cont_periodo = 63 then
           cont_periodo <= 0;
        else
            cont_periodo <= cont_periodo + 1;
        end if;
    end if;
end process;

process(reset, clk)
begin
    if reset = '1' then
        cont_periodo_aux <= 0;
        cont_periodo_producto <= 0;
        cont_periodo_int <= 0;
    elsif clk'event and clk = '1' then
        cont_periodo_aux <= cont_periodo;
        cont_periodo_producto <= cont_periodo_aux;
        cont_periodo_int <= cont_periodo_producto;
    end if;
end process;
```

Figura 55. Fragmento de código que solventa el problema de los retrasos en las operaciones.

Si la señal **cont_periodo** fuese la que controlase todas las operaciones, se perderían datos en la señal **cont_x** debido a los retardos (Figura 56). Al utilizar las señales registradas, se solventa el problema como se observa en la Figura 57. Obsérvese en esta última figura que cuando la señal **cont_periodo** llega al final de la cuenta, se obtienen tres datos más que en el otro caso en lugar de resetearse, ya que está controlado por la señal registrada **cont_periodo_int**.

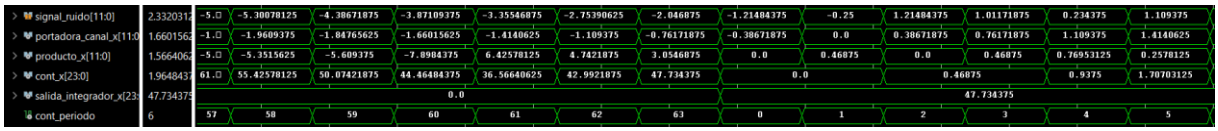


Figura 56. Simulación del problema de las operaciones al controlarlo únicamente con la señal ***cont_periodo***

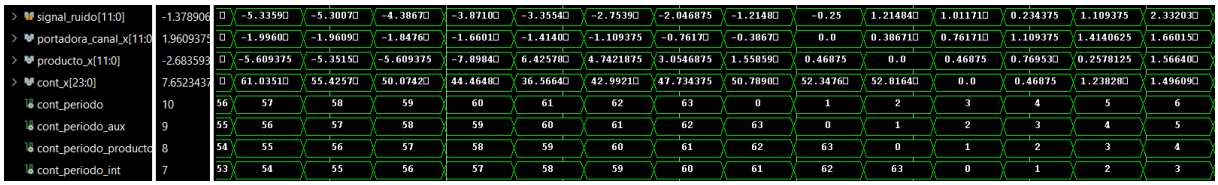


Figura 57. Simulación de la solución del problema de las operaciones utilizando las señales registradas

En la Figura 58 y Figura 59 se observa el proceso de multiplicación y extracción de la parte central del resultado y el proceso de suma acumulativa que realiza el integrador. Obsérvese en esta última que para el caso de las modulaciones ASK y BPSK, el canal *y* se pone a cero ya que no hay componente en cuadratura.

```
process(clk, reset, cont_periodo_producto)
begin
    if(reset = '1') then
        producto_x <= (others => '0');
        producto_y <= (others => '0');

    elsif clk'event and clk = '1' then
        if cont_periodo_producto = 63 then
            producto_x <= (others => '0');
            producto_y <= (others => '0');
        else
            producto_x <= extraer(signal_modulada * portadora_canal_x);
            producto_y <= extraer(signal_modulada * portadora_canal_y);
        end if;
    end if;
end process;
```

Figura 58. Fragmento del código que realiza la operación de multiplicación y extrae la parte central del resultado

```
process(reset, clk, modulacion, cont_periodo_int)
     begin
        if(reset = '1') then
            cont_x <= (others => '0');
            cont_y <= (others => '0');
        elsif clk'event and clk = '1' then
                if cont_periodo_int = 63 then
                    cont_x <= (others => '0');
                    cont_y <= (others => '0');
                elsif (modulacion = "00000010") or (modulacion = "00000011") then -- FSK y QPSK
                    cont_x <= cont_x + producto_x;
                    cont_y <= cont_y + producto_y;
                else                                                              -- ASK Y BPSK (no se usa el canal y)
                    cont_x <= cont_x + producto_x;
                    cont_y <= (others => '0');
                end if;
        end if;
end process;
```

Figura 59. Fragmento del código del integrador

Por último, se encuentra el proceso del decisor (Figura 60) en el que para todos los casos se obtienen las constelaciones guardando el último valor se la señal del integrador (es decir, cuando se llegue al final de la cuenta, antes del reseteo) y se genere la señal demodulada en función de los valores umbrales que se han especificado según el tipo de modulación. Por ejemplo, para el caso de la modulación ASK, el valor umbral se obtuvo observando en simulación el valor medio de la señal del integrador. Lo mismo ocurre con la modulación FSK, pero utilizando ambos canales. Además, como en el caso de la modulación BPSK el integrador toma valores positivos y negativos, el punto medio es cero. Ocurre algo similar con la modulación QPSK, pero en este caso se realiza para los dos canales obteniendo así cuatro casos diferentes.

```
process(clk, reset, cont_periodo_int, cont_periodo, cont_x, cont_y,  modulacion) -- DECISOR
    begin
       if(reset = '1') then
           signal_dem <= (others => '0');
           error <= '0';
           salida_integrador_x <= (others => '0');
           salida_integrador_y <= (others => '0');

       elsif clk'event and clk = '1' then

           if modulacion = "00000000" then -- ASK
               if (cont_periodo_int = 63) then
                   salida_integrador_x <= cont_x;
                   salida_integrador_y <= cont_y;
               end if;
               if (cont_periodo = 63) then
                   if cont_x > 16336 then
                       signal_dem <= (others => '1');
                   else
                       signal_dem <= (others => '0');
                   end if;
               end if;

           elsif modulacion = "00000001" then -- BPSK
               if (cont_periodo_int = 63) then
                   salida_integrador_x <= cont_x;
                   salida_integrador_y <= cont_y;
               end if;
               if (cont_periodo = 63) then
                   if cont_x > 0 then
                       signal_dem <= (others => '1');
                   else
                       signal_dem <= (others => '0');
                   end if;
               end if;
```

```
           elsif modulacion = "00000010" then -- FSK
               if (cont_periodo_int = 63) then
                   salida_integrador_x <= cont_x;
                   salida_integrador_y <= cont_y;
               end if;
               if (cont_periodo = 63) then
                   if cont_x > 16536 then
                       signal_dem <= (others => '1');
                       error <= '0';
                   elsif cont_y > 15678 then
                       signal_dem <= (others => '0');
                       error <= '0';
                   else
                       signal_dem <= (others => '0');
                       error <= '1';
                   end if;
               end if;

           elsif modulacion = "00000011" then -- QPSK
               if (cont_periodo_int = 63) then
                   salida_integrador_x <= cont_x;
                   salida_integrador_y <= cont_y;
               end if;
               if (cont_periodo = 63) then
                   if ((cont_x >= 0) and (cont_y >= 0)) then
                       signal_dem <= "00";
                       error <= '0';
                   elsif (cont_x < 0) and (cont_y >= 0) then
                       signal_dem <= "01";
                       error <= '0';
                   elsif ((cont_x < 0) and (cont_y < 0)) then
                       signal_dem <= "11";
                       error <= '0';
                   elsif (cont_x >= 0) and (cont_y < 0) then
                       signal_dem <= "10";
                       error <= '0';
                   else
                       error <= '1';
                       signal_dem <= "11";
                   end if;
               end if;
```

Figura 60. Fragmento del código que muestra la parte del decisor que controla las modulaciones y el reseteo de las señales

En la Figura 61, Figura 62, Figura 63 y Figura 64 se muestran las simulaciones del demodulador para los diferentes tipos de modulaciones, pudiéndose observar el valor de la señal demodulada en **signal_dem**, las rampas generadas por el integrador en las señales **cont_x** y **cont_y** y el valor de las constelaciones, es decir, el último valor que toma el integrador en las cuatro últimas señales de las figuras.

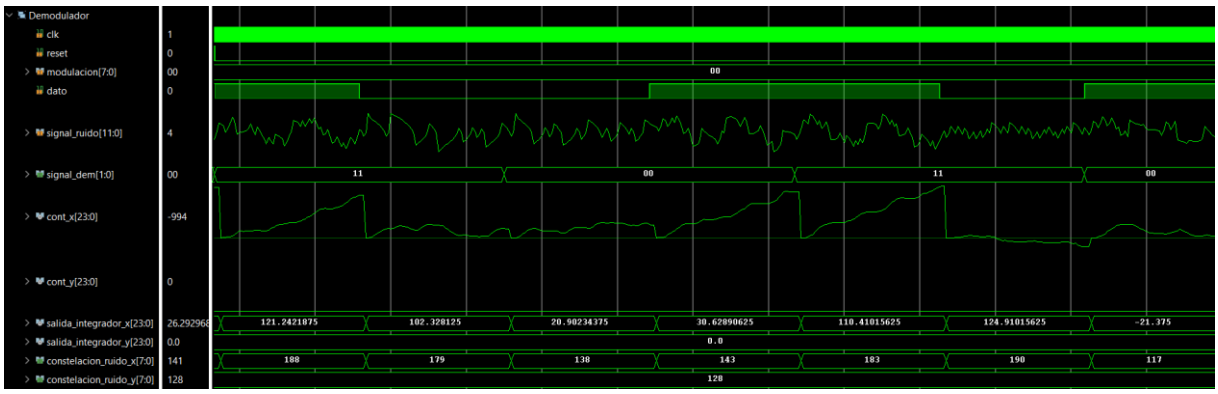


Figura 61. Simulación de la demodulación de una señal modulada en ASK

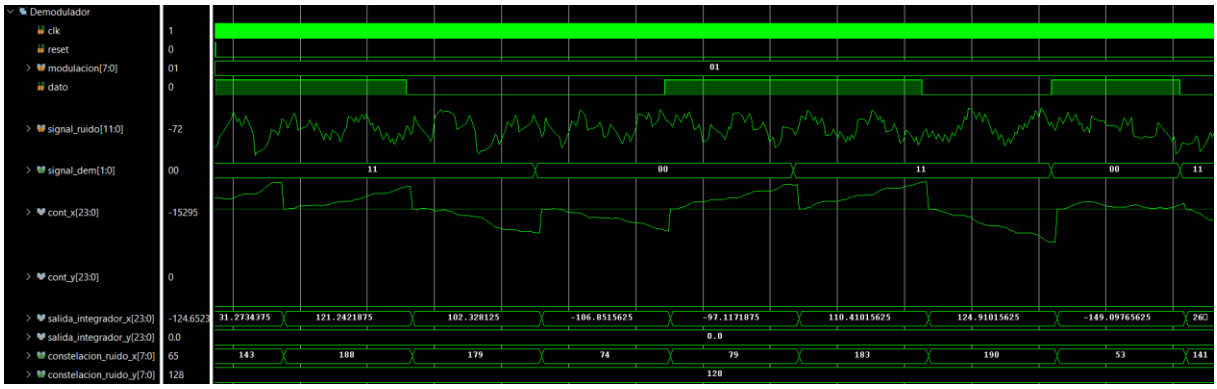


Figura 62. Simulación de la demodulación de una señal modulada en BPSK

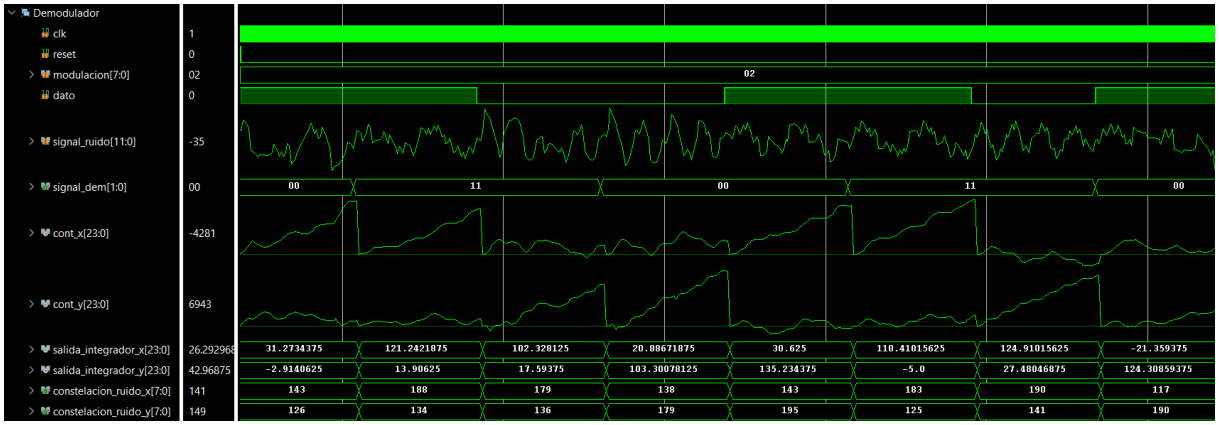


Figura 63. Simulación de la demodulación de una señal modulada en FSK

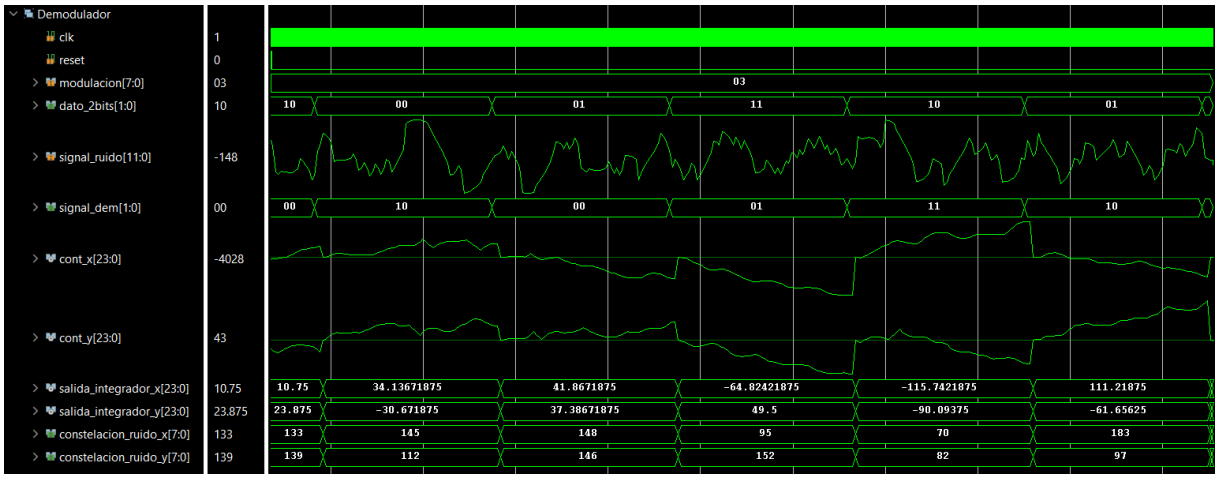


Figura 64. Simulación de la demodulación de una señal modulada en QPSK

Se extrayeron los ocho bits MSB de las señales de la salidas de los canales del integrador y las constelaciones con el fin de envíarlos por la UART, ya que para operar se utilizaron más de ocho bits.

4.1.7. Módulo del divisor de frecuencia de las tramas

Inicialmente se diseñó la señal de reloj que controlaba el envío de tramas, mostrado en la Figura 65, utilizando el valor mínimo de cuenta necesaria (8680 cuentas como se vio en el capítulo 3) y lo que sucedía era que, al observar los resultados en *LabView*, en las señales se observaban ciertas perturbaciones. Por ello, se decidió aumentar esta cuenta, es decir, aumentar el tiempo de envío entre tramas y se solucionó este problema.

```
P_DivFreq: Process(reset,clk)
begin
    if reset = '1' then
        cont_baud <= (others => '0');
        clk_aux <= '0';
    elsif clk'event and clk='1' then
            if cont_baud = 50000-1 then
                clk_aux <= not(clk_aux);
                cont_baud <= (others => '0');
            elsif cont_baud = (50000/2) - 1 then
                clk_aux <= not(clk_aux);
                cont_baud <= cont_baud + 1;
            else
                cont_baud <= cont_baud + 1;
            end if;
    end if;
end process;

clk_trama <= clk_aux;
```

Figura 65. Fragmento de código del divisor de frecuencia de una trama

Para la señal periódica que indica el envío de todos los paquetes, la cuenta a la que debe llegar el contador es el número total de paquetes, es decir, 12 cuentas, controlado por la señal **clk_trama** obtenida en el anterior proceso (Figura 66).

```
P_DivFreq_Tramas: Process(reset,clk_aux)
begin
    if reset = '1' then
        cont_baud_tramas <= (others => '0');
        clk_aux_tramas <= '1';
    elsif clk_aux'event and clk_aux='1' then
            if cont_baud_tramas = 11 then
                clk_aux_tramas <= not(clk_aux_tramas);
                cont_baud_tramas <= (others => '0');
            elsif cont_baud_tramas = 5 then
                clk_aux_tramas <= not(clk_aux_tramas);
                cont_baud_tramas <= cont_baud_tramas + 1;
            else
                cont_baud_tramas <= cont_baud_tramas + 1;
            end if;
    end if;
end process;

clk_trama_x12 <= clk_aux_tramas;
```

Figura 66. Fragmento de código del divisor de frecuencia para los paquetes

4.1.8. Módulo de control de envío

Para controlar el número de paquetes que se están enviando, se ha diseñado el contador de la Figura 67 controlado por la señal periódica **clk_trama**.

```
process (clk_trama, reset)
begin
    if reset = '1' then
        cont <= (others => '0');
    elsif clk_trama'event and clk_trama='1' then
        if cont = 11 then
            cont <= (others => '0');
        else
            cont <= cont + 1;
        end if;
    end if;
end process;
```

Figura 67. Fragmento del código del conteo de los paquetes que se van enviando

Algunas señales que se obtienen del resto de módulos del proyecto presentan más de 8 bits, que es la cantidad máxima de bits que se pueden enviar en una sola trama. Por ello, los 8 bits MSB se han convertido en complemento a 2 para su correcto envío por la UART, como se muestra en la Figura 68.

```
signal_modulada_unsigned <= signal_modulada(11 downto 4) + conv_signed(2**(8-1),8);
canal_x_unsigned <= canal_x + conv_signed(2**(8-1),8);
canal_y_unsigned <= canal_y + conv_signed(2**(8-1),8);
signal_ruido_unsigned <= signal_ruido(11 downto 4) + conv_signed(2**(8-1),8);
ruido_unsigned <= ruido(11 downto 4) + conv_signed(2**(8-1),8);
```

Figura 68. Fragmento del código de la conversión de las señales a 8 bits

Para el envío de una trama, la señal **transmite** debe estar activa únicamente un ciclo de reloj. Para ello, se ha diseñado un detector de flanco (Figura 69) para la señal **clk_trama**, de manera que la señal **transmite** se active en el flanco de bajada de la misma, registrando el valor del paquete justo en la mitad del tiempo de envío.

```
P_BIEST: process (reset, clk)
begin
    if reset = '1' then
        B <= '0';
        Z <= '0';
    elsif clk'event and clk = '1' then
        B <= clk_trama;
        Z <= B;
    end if;
end process;

FlancoUp <= '1' when Z = '1' and B = '0' else '0';

process(FlancoUp, transmitiendo)
begin

    if FlancoUp = '1' then
        transmite <= '1';
    elsif transmitiendo = '1' then
        transmite <= '0';
    end if;

end process;
```

*Figura 69. Fragmento del código del detector de flanco para la señal **transmite***

Según el valor de la cuenta, se van enviando los datos a través de la UART de manera ordenada como se muestra en la Figura 70.

```
process (clk, reset)
begin
    if reset = '1' then
        datos_8bits <= (others => '0');
    elsif clk'event and clk='1' then
        if transmitiendo = '0' then
            if cont = 0 then                        -- BYTE DE INICIO (FF)
                datos_8bits <= "11111111"; --255

            elsif cont = 1 then                     -- SEÑAL MODULADA
                datos_8bits <= signal_modulada_unsigned;

            elsif cont = 2 then                      -- SEÑAL DE DATOS
                if modulacion = "00000011" then -- QPSK (2 bits)
                    if dato_2bits = "11" then
                        datos_8bits <= "11110000"; -- 240
                    elsif dato_2bits = "10" then
                        datos_8bits <= "10100000"; -- 160
                    elsif dato_2bits = "01" then
                        datos_8bits <= "01010000"; -- 80
                    else
                        datos_8bits <= (others => '0');
                    end if;
                else                                -- Resto de modulaciones (1 bit)
                    if dato = '1' then
                        datos_8bits <= "11110000"; -- 240
                    else
                        datos_8bits <= (others => '0');
                    end if;
                end if;

            elsif cont = 3 then                      -- SEÑAL DEMODULADA
                if signal_dem = "11" then
                    datos_8bits <= "11110000"; -- 240
                elsif signal_dem = "10" then
                    datos_8bits <= "10100000"; -- 160
                elsif signal_dem = "01" then
                    datos_8bits <= "01010000"; -- 80
                else
                    datos_8bits <= (others => '0'); -- 0
                end if;
```

```
            elsif cont = 4 then                     -- CANAL X
                datos_8bits <= canal_x_unsigned;

            elsif cont = 5 then                     -- CANAL Y
                datos_8bits <= canal_y_unsigned;

            elsif cont = 6 then                     -- SEÑAL MODULADA CON RUIDO
                datos_8bits <= signal_ruido_unsigned;

            elsif cont = 7 then                     -- SEÑAL DE RUIDO
                datos_8bits <= ruido_unsigned;

            elsif cont = 8 then                     -- CONSTELACION X SIN RUIDO
                datos_8bits <= constelacion_x;

            elsif cont = 9 then                     -- CONSTELACION Y SIN RUIDO
                datos_8bits <= constelacion_y;

            elsif cont = 10 then                     -- CONSTELACION X CON RUIDO
                datos_8bits <= constelacion_ruido_x;

            elsif cont = 11 then                     -- CONSTELACION Y CON RUIDO
                datos_8bits <= constelacion_ruido_y;
            end if;
        end if;
    end if;
```

Figura 70. Fragmento del código que muestra el control del envío de paquetes y el reseteo.

Obsérvese en la Figura 71 que los datos se almacenan en la señal **datos_8bits** a cada flanco de subida de la señal **clk_trama** y se envían cuando **transmite** se encuentra activa, es decir, a la mitad de cada ciclo de esta señal, evitando enviar los bits cerca de las transiciones donde se puede tomar el valor del bit contiguo. También se puede observar el orden de los paquetes siendo el valor 255 la trama de inicio.

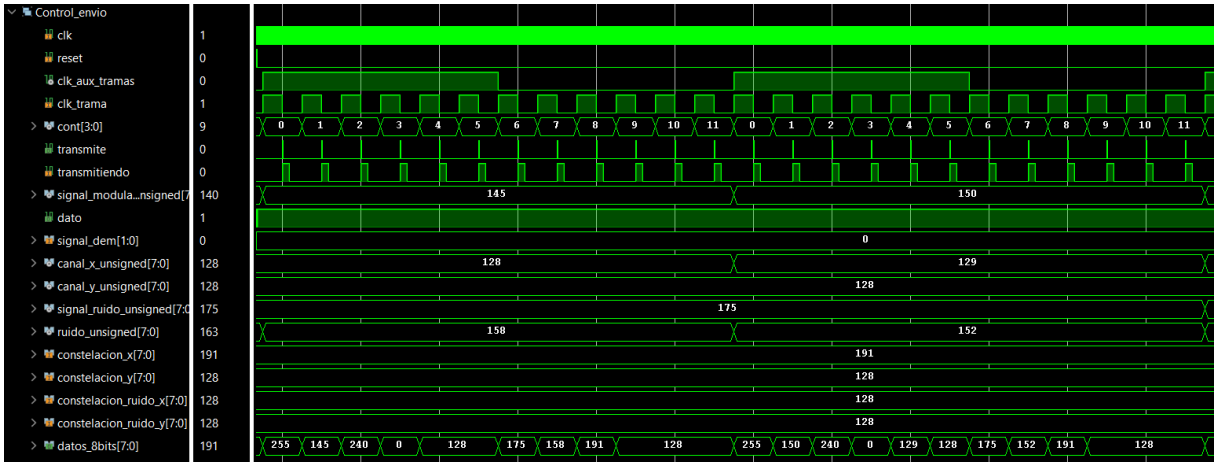


Figura 71. Simulación del envío de los paquetes

4.2. Resultados en *LabView*

A continuación, se muestran los resultados que se visualizan en el entrenador de comunicaciones en *LabView*. Como se ha comentado, se puede intercambiar de modulación cuando el usuario lo requiera entre ASK, BPSK, FSK y QPSK. En este entorno, se pueden visualizar las señales del bloque transmisor y receptor, pudiéndose observar los bits de datos, la señal modulada y las constelaciones. Para el caso del receptor, también se puede observar los canales del integrador.

La visualización de las señales del entrenador en *LabView* se muestra en la Figura 72, Figura 73, Figura 74 y Figura 75. Las gráficas superiores se corresponden a las señales del transmisor mientras que las inferiores a las del receptor. En el margen izquierdo se observa las constelaciones de los símbolos de cada modulación. Cabe destacar que los puntos de las constelaciones en el receptor presentan un ligero desplazamiento debido al ruido en la señal recibida. Además, en las gráficas de la derecha se muestran en color azul oscuro la señal modulada, en color rojo los datos tanto transmitidos como demodulados, y en color verde y azul claro los canales *x* e *y* para el caso del receptor. Además, también se observa el desplegable para cambiar el tipo de modulación y el número de muestras a visualizar situados en la parte superior de la pantalla.

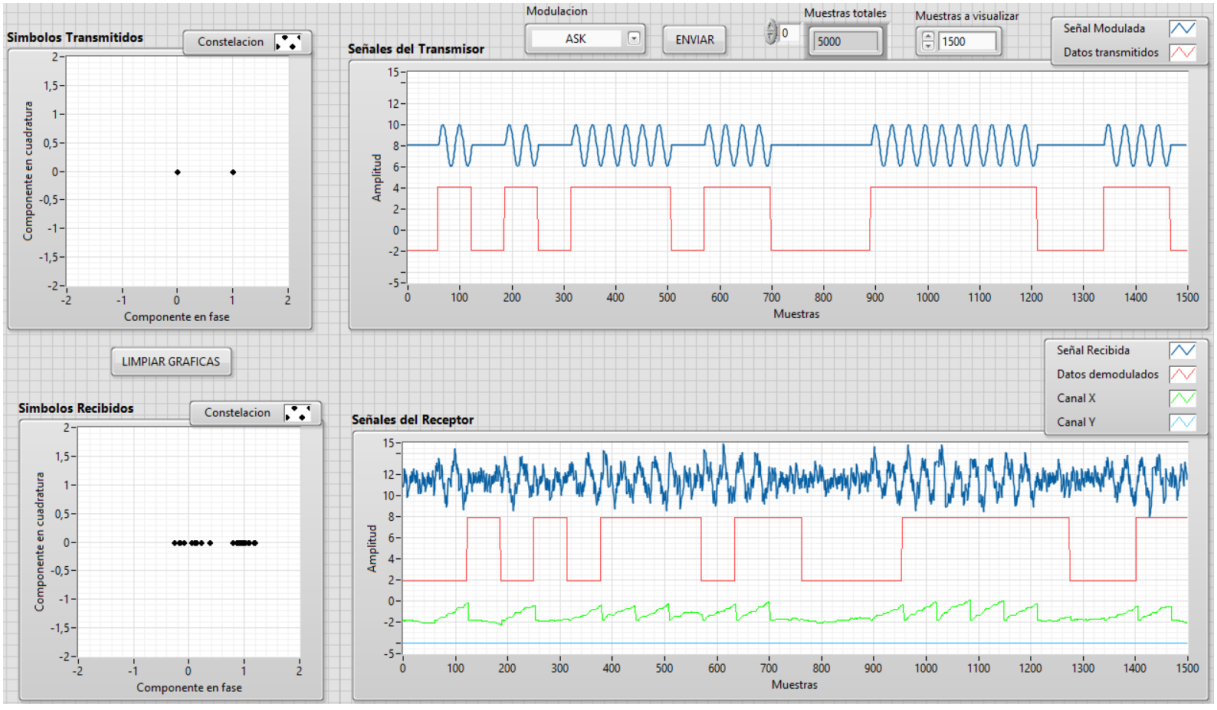


Figura 72. Visualización en LabView del entrenador con la modulación ASK seleccionada

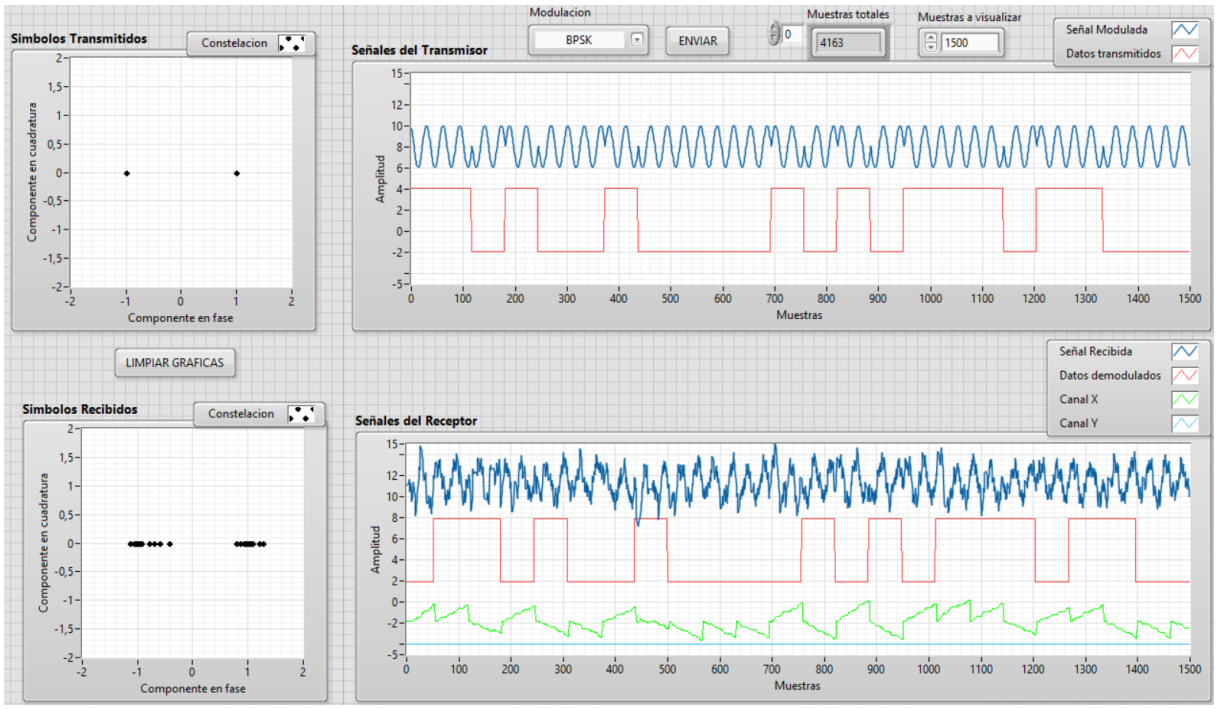


Figura 73. Visualización en LabView del entrenador con la modulación BPSK seleccionada

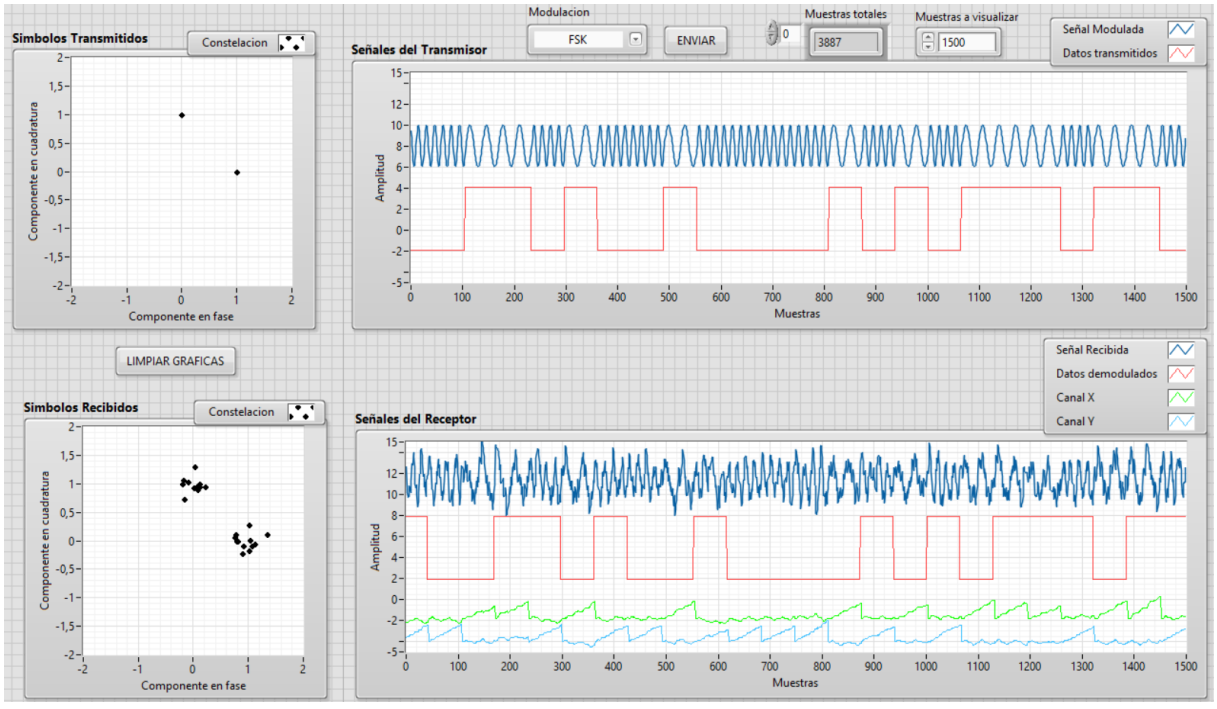


Figura 74. Visualización en LabView del entrenador con la modulación FSK seleccionada.

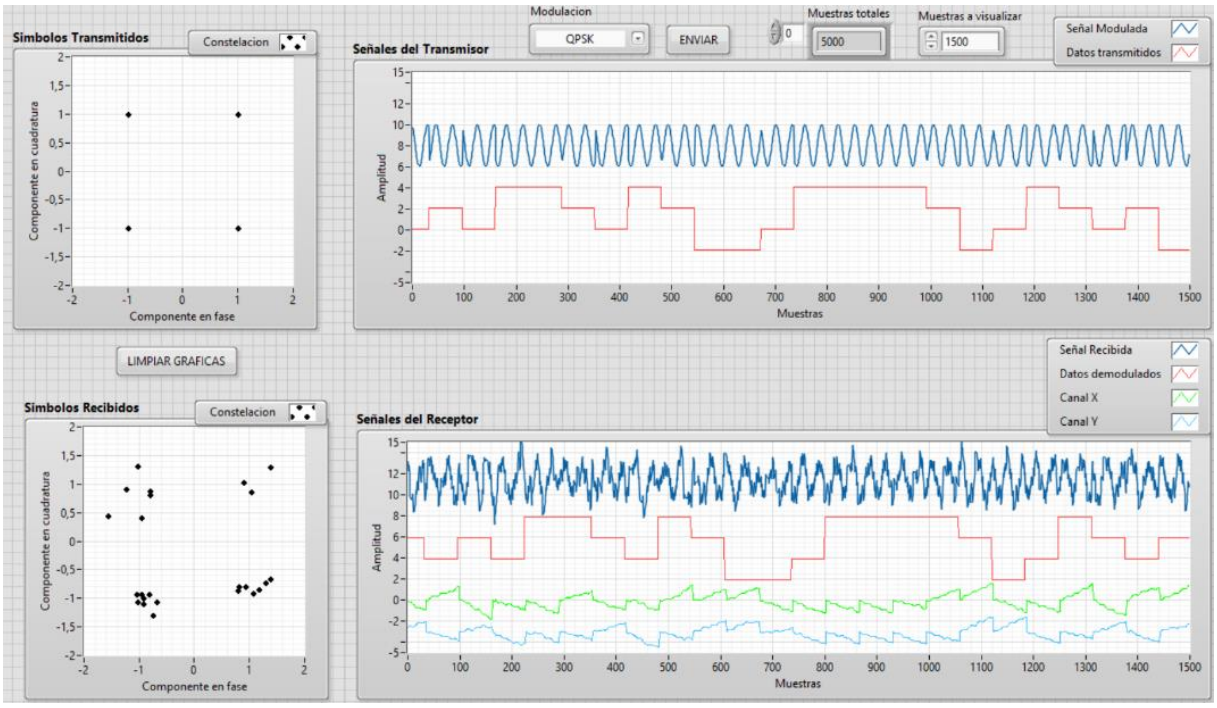


Figura 75. Visualización en LabView del entrenador con la modulación QPSK seleccionada.

Además, presenta un histograma que indica la frecuencia en la que las muestras se repiten, dibujando una distribución de probabilidad gaussiana. En función de la posición de los *switches* de la FPGA, se va cambiando la ganancia de la señal de ruido teniendo hasta 15 ganancias de ruido. En la Figura 76 se muestra la distribución de ruido a la máxima ganancia posible.

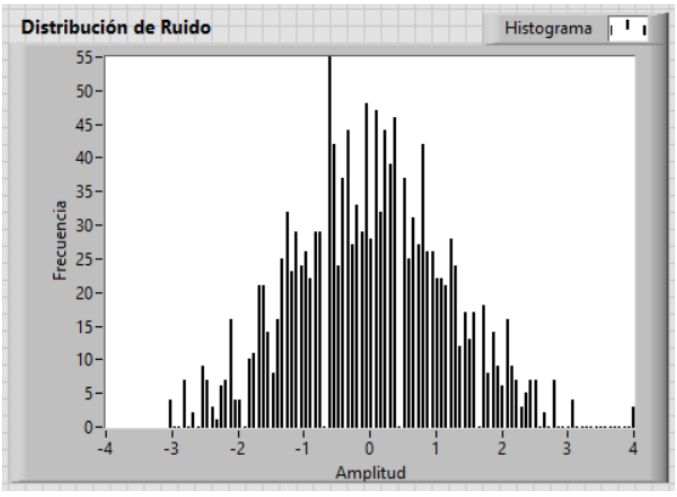


Figura 76. Visualización en LabView de la distribución del ruido cuando la posición de los switches se encuentran todas en alta

4.3. Adaptación del código antiguo a la nueva FPGA

La tarjeta de desarrollo que se utilizó para el diseño del entrenador de comunicaciones digitales utilizado en las prácticas de la asignatura 'Sistemas de Comunicación' del Máster de Ingeniería Industrial (especialidad Electrónica) fue la *Spartan-3A/3AN Starter Kit Board*, mostrada en la Figura 77. Esta tarjeta presenta un reloj de 50 MHz e incluye un puerto RS-232 para la comunicación a través de un módulo UART y un convertidor digital a analógico (DAC, *Digital-to-Analog Converter*) serie de cuatro canales y compatible con el bus SPI (*Serial Peripheral Interface*). El SPI es un bus síncrono *full-duplex* orientado a carácter que utiliza únicamente cinco líneas de comunicación.

Figura 77. Placa de desarrollo Spartan-3A/3AN Starter Kit Board (fuente: https://dynamoelectronics.com/tienda/spartan-3e-starter-board-xc3s500e/)

El entrenador de comunicaciones digitales de esa asignatura da la posibilidad de estudiar diferentes técnicas de modulación digital tanto binarias (ASK, BPSK y FSK) como de niveles múltiples (QPSK, 8-PSK, 16-PSK y 16-QAM), analizando sus formas de onda y su rendimiento frente al ruido. Las señales se observan en el entorno *LabView* y también es posible observar alguna de ellas en un osciloscopio con el uso del módulo DAC que posee la FPGA.

Para cada modulación, este entrenador permite visualizar tanto las señales del transmisor, como son la señal modulada sin ruido y la señal de datos; como las señales del receptor, que son la señal modulada con ruido, la señal demodulada y los canales *X* e *Y* del integrador (Figura 78). Además, en ambos, transmisor y

receptor, se pueden observar las constelaciones y la distribución de ruido activando el cambio de panel (Figura 79). Cabe destacar que es posible cambiar la amplitud de ruido mediante este *software*.

Además, este entrenador permite la introducción de retardos para observar el efecto que producen sobre la demodulación de datos y también el cálculo del número de errores en la demodulación de los símbolos recibidos, pudiendo así obtener la tasa de error de bit (BER).

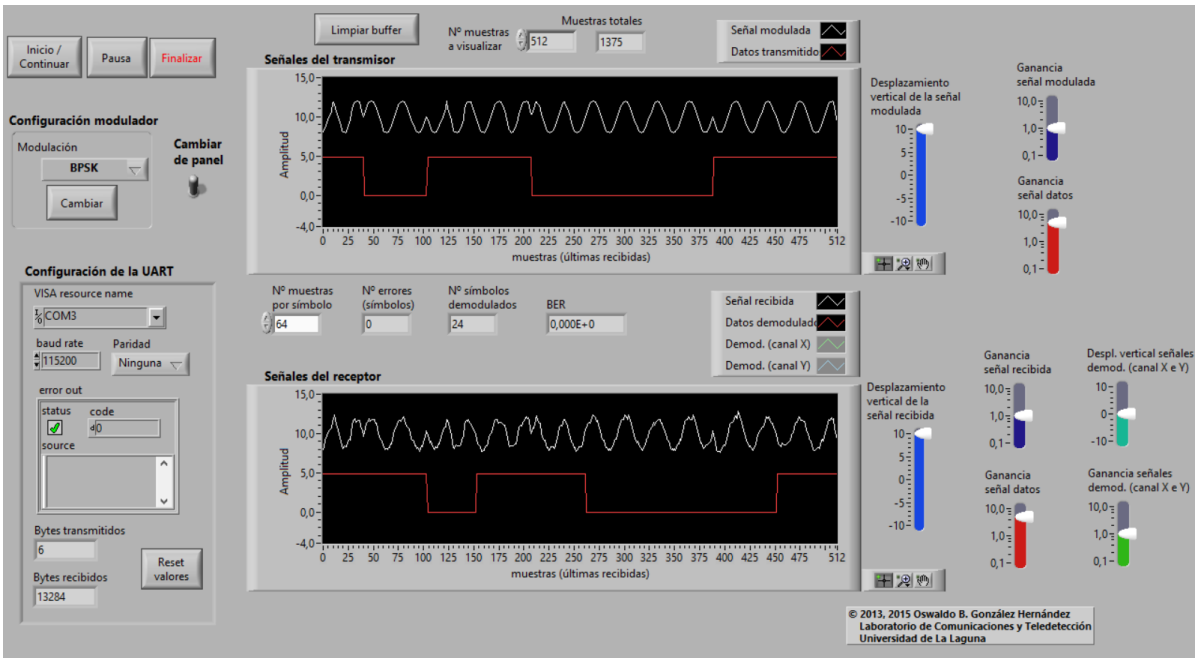


Figura 78. Panel principal del software del entrenador de comunicaciones de la asignatura Sistemas de Comunicación.

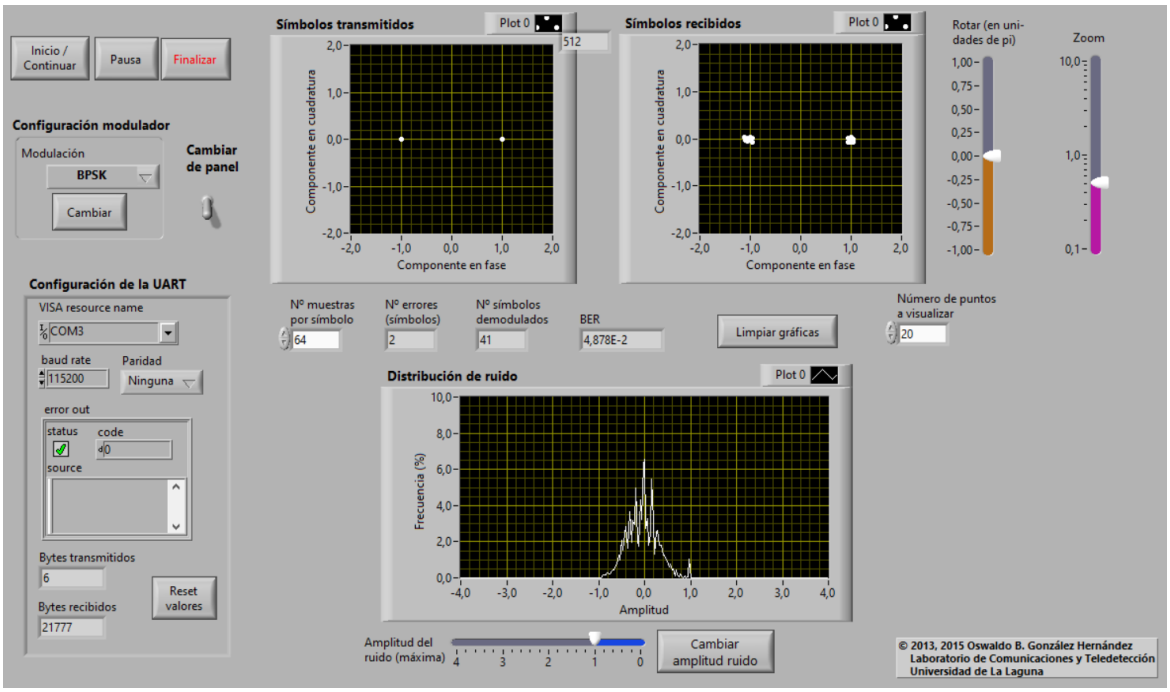


Figura 79. Panel secundario del software del entrenador de comunicaciones de la asignatura Sistemas de Comunicación.

Para adecuar el código del entrenador a la tarjeta *Nexys A7*, lo primero que se tuvo en cuenta es el cambio de la frecuencia de reloj. Esta tarjeta presenta una frecuencia de la señal de reloj de 100 MHz mientras que la tarjeta *Spartan-3A/3AN* tiene un reloj de 50 MHz, por lo que se utilizó un divisor de frecuencia, mostrado en la Figura 80, para obtener un reloj de 50 MHz a fin de que cada módulo del proyecto funcionase correctamente.

```
-- Divisor de frecuencia de 100Mhz a 50Mhz
    Div_Freq50MHz: process(reset,reloj)
    begin

        if (reset = '1') then
            clk_50MHz <= '0';
            cont_clk <= (others => '0');

        elsif reloj'event and reloj = '1' then

            if (cont_clk = 1) then
                clk_50MHz <= '0';
                cont_clk <= (others => '0');
            else
                clk_50MHz <= '1';
                cont_clk <= cont_clk + 1;
            end if;

        end if;
    end process;
```

Figura 80. Divisor de frecuencia para pasar de 100 MHz a 50 MHz

Una vez cambiada la frecuencia de reloj, se modificaron los pines de la placa. El inconveniente de la nueva FPGA es que no presenta un DAC, por lo que los pines asignados a ese módulo se han conectado a los pines del conector PMOD para que, en un futuro, se introduzca un módulo DAC y se pueda implementar esta función.

Para la introducción de los retardos de utilizaron los cuatro primeros *switches* mientras que el último corresponde al *reset*. Luego, el valor obtenido de la UART desde el ordenador se muestra en los 8 últimos *leds* de la placa. El resto de los pines que se asignaron fueron la línea de transmisión *Tx* y recepción *Rx* que se conectaron a los pines correspondientes.

4.4. Recursos usados

En este apartado se exponen los recursos de la FPGA utilizados por ambos entrenadores, divididos en LUT (*Look-Up Table*), LUTRAM, FF (*Flip-Flop*), DSP (*Digital Signal Processor*), IO (*Input/Output*) y BUFG (*Global Clock Simple Buffer*).

Las LUT son pequeñas SRAM asíncronas de solo lectura utilizadas para implementar la lógica combinacional y únicamente se pueden modificar durante la configuración de la FPGA. Sin embargo, normalmente, la mitad de las LUT se pueden escribir, por lo que se pueden utilizar para implementar muchas RAM pequeñas (“RAM distribuidas”) [1].

Por otro lado, se encuentran los FF que son celdas de memoria de un solo bit utilizados para mantener el estado, cuyo objetivo principal es escribir en memoria.

La serie 7 de *FPGA* de *Xilinx* se encuentra constituida por varios DSP, que son elementos lógicos configurables dedicados a la multiplicación y a la suma. Estos combinan gran velocidad de funcionamiento con un tamaño reducido y mejoran la velocidad y eficiencia de muchas tareas, no solo del procesado digital de señal, sino, por ejemplo, de la generación de direcciones de memoria [10].

Por último, las IO se corresponden a las entradas y salidas del proyecto mientras que BUFG es un búfer global y su entrada es la salida de IBUFG, otro búfer global de primer nivel conectado al pin de entrada del reloj global dedicado. Todas las señales de entrada de los pines del reloj global deben pasar a través de esta unidad.

Resource	Utilization	Available	Utilization %
LUT	552	63400	0.87
FF	449	126800	0.35
DSP	4	240	1.67
IO	8	210	3.81
BUFG	2	32	6.25

Figura 81. Utilización de recursos del entrenador de comunicaciones digitales propio en la tarjeta Artix-7

Resource	Utilization	Available	Utilization %
LUT	2771	63400	4.37
FF	330	126800	0.26
DSP	53	240	22.08
IO	21	210	10.00
BUFG	6	32	18.75

Figura 82. Utilización de recursos del entrenador de comunicaciones digitales de la asignatura 'Sistemas de Comunicación' en la tarjeta Artix-7

Como se puede observar en las Figura 81 y Figura 82, el entrenador de comunicaciones digitales diseñado presenta menos recursos utilizados que el entrenador de la asignatura 'Sistemas de Comunicación'. Esto es debido a que este último implementa un mayor número de funciones, tipos de modulación seleccionables y, por tanto, también un mayor número de demoduladores implementados.

El consumo de potencia de las FPGA viene dado por dos componentes principales. Por un lado, la potencia estática debida a las corrientes de fugas de las capacidades parásitas de los transistores de la FPGA. Estos transistores utilizan tecnología CMOS. Por tanto, con los avances en la tecnología, los tamaños de los transistores se reducen y este consumo puede ser mucho más relevante, llegando a ser mayor que el consumo dinámico.

Las FPGA de RAM estática guardan su programación en *Static* RAM (SRAM), lo que implica dos consumos importantes. Uno de ellos es el consumo transitorio inicial debido a la programación y el otro es el consumo de todas las celdas SRAM, las cuales consumen energía debido al refresco de la memoria [6].

Por otro lado, la potencia dinámica se corresponde a la potencia debida al modelo de entrada de datos y la actividad interna de los nodos. Depende de los niveles de tensión, la lógica usada y los recursos de interconexión utilizados por lo que esto hace que dicha potencia dependa del circuito digital implementado. Incluye la corriente estática de las terminaciones I/O, *clock managers* y otros circuitos que necesitan potencia mientras son utilizados [9].

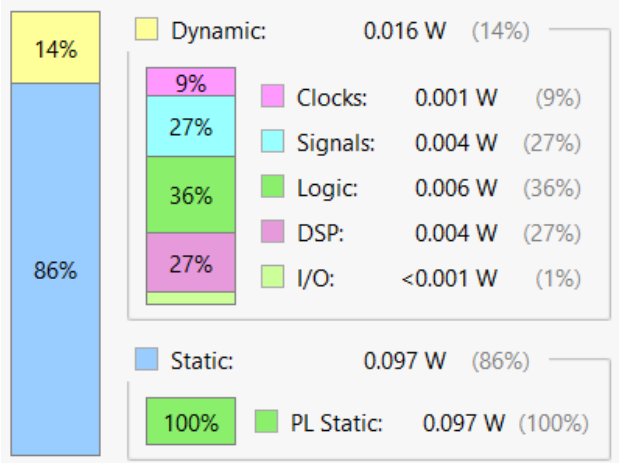


Figura 83. Potencia "On-Chip" del entrenador de comunicaciones digitales propio

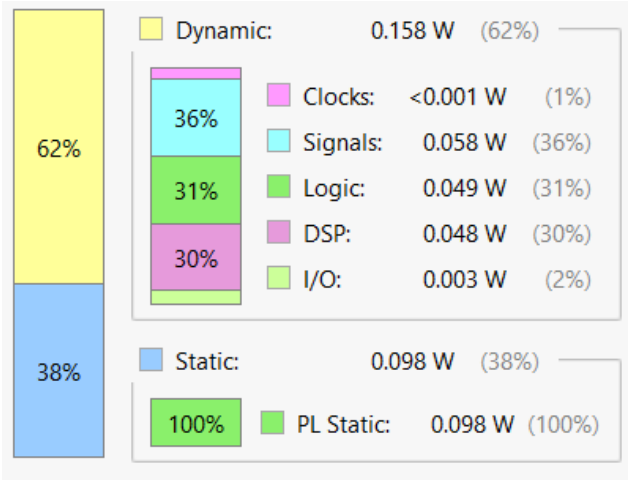


Figura 84. Potencia "On-Chip" del entrenador de comunicaciones digitales de la asignatura Sistemas de Comunicación

Podemos observar en las Figura 83 y Figura 84 que el entrenador de comunicaciones digitales diseñado presenta un consumo de potencia dinámico muy inferior al entrenador de la asignatura de 'Sistemas de Comunicación'. Esto es debido a la complejidad del circuito digital implementado, ya que como se observó anteriormente, este último entrenador utiliza muchos más recursos.

Capítulo 5. Presupuesto

Capítulo 5. Presupuesto

En este capítulo podemos ver un desglose de los costes del presente proyecto, tanto de materiales como de mano de obra, constituyendo el presupuesto total necesario para llevar a cabo el mismo. Se incluyen además los gastos generales y el beneficio industrial.

COSTES MATERIALES			
Concepto	**Cantidad**	**Precio unitario (€)**	**Coste total (€)**
Tarjeta de desarrollo Nexys A7	1	299,92	299,92
Licencia Vivado 2020.1	1	642,00	642,00
Licencia LabView 2011	1	489,00	489,00
	Total materiales		**1430,92**

COSTES MANO DE OBRA			
Concepto	**Tiempo (h)**	**Precio hora (€/h)**	**Coste total (€)**
Búsqueda de información	60	10,00	600,00
Diseño y análisis	100	20,00	2000,00
Simulaciones	60	15,00	900,00
Implementación	40	20,00	800,00
Documentación	40	10,00	400,00
	Total mano de obra		**4700,00**

COSTES TOTAL DEL PROYECTO	
Concepto	**Coste total (€)**
Materiales	1430,92
Mano de obra	4700,00
Gastos generales (6%)	367,86
Beneficio industrial (13%)	797,02
Total costes	**7295,80**

Capítulo 6. Conclusiones

Capítulo 6. Conclusiones

6.1. Conclusiones

En este Trabajo Fin de Máster se ha realizado satisfactoriamente el diseño de un entrenador de comunicaciones basado en FPGA y la visualización de resultados en *LabView*. En este entrenador se han podido visualizar las formas de onda que se encuentran en el transmisor y receptor de las modulaciones ASK, BPSK, FSK y QPSK.

Por otro lado, también se ha adaptado el código del entrenador de comunicaciones de la asignatura 'Sistemas de Comunicación' implementado en la tarjeta de desarrollo *Spartan-3A/3AN Starter Kit Board* y diseñado en el entorno ISE, a las nuevas FPGA *Nexys A7* utilizando el entorno *Vivado*.

Los datos de ambos entrenadores se han visualizado en el software *LabView*, en el cual se comprobó que las señales obtenidas eran las correctas para cada modulación. Por tanto, se concluye que se han cumplido todos los objetivos propuestos para este proyecto.

Estos entrenadores de comunicaciones ofrecen una facilidad de comprensión sobre el mundo de los sistemas de comunicación al visualizar en tiempo real las señales que se generan en cada tipo de modulación, dejando a un lado los modelos matemáticos e ilustraciones de los libros. Es por ello que supone una gran herramienta didáctica destinada a la adquisición de conocimientos de una forma más visual.

Además, ofrece la ventaja de ser económico, por lo que se puede adaptar a las necesidades de cualquier entidad o individuo que tenga como objetivo impartir este tipo de conocimientos a un bajo coste.

6.2. Conclusions

In this master's Thesis, the design of a communications training system based on FPGA and visualisation of the results in *LabView* has been successfully carried out. This training system allows us to visualise the waveforms at the transmitter and receiver ends for ASK, BPSK, FSK and QPSK modulations.

On the other hand, the training system developed for the subject 'Communication Systems', which was originally implemented on the *Spartan-3A/3AN Starter Kit Board* and designed in the ISE environment, has also been adapted by using the *Vivado* environment and implemented in the newer FPGA family *Nexys A7*.

The data from both training systems have been visualised in *LabView* software, where the signals obtained were verified to be correct for each modulation. Therefore, it is concluded that all the objectives proposed for this project have been achieved.

These training systems offer an easy understanding of the world of communication systems by visualising in real time the signals generated for each type of modulation, leaving aside mathematical models and illustrations in books. Therefore, it constitues a great didactic tool for the acquisition of knowledge about digital communication systems in a more visual way.

In addition, it offers the advantage of being economical, so it can be adapted to the needs of any entity or individual whose objective is to teach this type of knowledge at a low cost.

6.3. Posibles líneas futuras

Como mejoras a este proyecto, se podrían añadir más modulaciones al entrenador de comunicaciones y visualizar las señales no sólo mediante el *software* de *LabView*, sino también a través del osciloscopio mediante la creación de un módulo DAC.

Por otro lado, se puede mejorar el intercambio de datos entre la FPGA y el *software LabView* para hacerla más rápida y con menos errores. Para ello, sería conveniente utilizar otros protocolos de comunicación, como el USB.

Por último, se podría implementar un módulo en el que los datos a enviar fuesen mediante un elemento real, por ejemplo, a través del micrófono integrado de la tarjeta *Nexys A7*, y observar en el receptor cómo se han obtenido esos datos a través de un altavoz.

Capítulo 7. Bibliografía

Capítulo 7. Bibliografía

[1] A. Brunete (2018). Sistemas Electrónicos Digitales) [Online]. Disponible: https://www.cartagena99.com/recursos/alumnos/apuntes/Apuntes_SED_FPGAs.pdf

[2] Alfke, P. “Efficient Shift Registers, LFSR Counters, and Long Pseudo Random Sequence Generators”. Application Note XAPP 052, Version 1.1, XILINX Corp., 1996.

[3] F. M. Sánchez y S. B. López, *Diseño de circuitos digitales con VHDL.* Madrid, 2010.

[4] F. M. Sánchez, S. B. López y C. R. Sánchez, *Diseño de sistemas digitales con VHDL.* Madrid, 2011.

[5] Holin, H. Harthong-Reeb analysis and digital circles. *The Visual Computer 8***,** pp. 8–17, 1991.

[6] J. P. Oliver, “Técnicas de bajo consumo en FPGAs”, Tesis doctoral, Universidad de La República, Montevideo, Uruguay, 2014.

[7] National Instruments (2022, 25 de mayo). Información General sobre NI-VISA [Online]. Disponible: https://www.ni.com/es-es/support/documentation/supplemental/06/ni-visa-overview.html

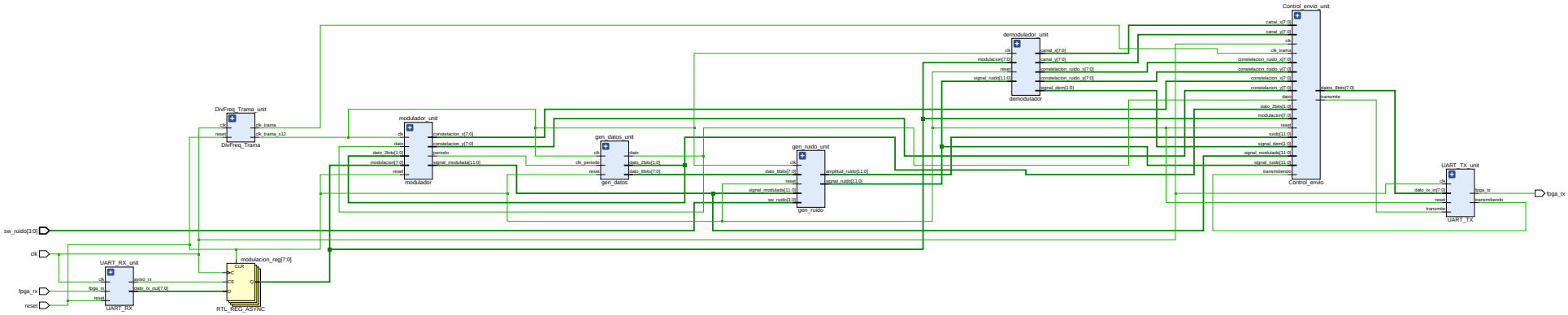
[8] National Instruments (2022, 22 de Julio). Shift Registers: Passing Values between Loop Iterations [Online]. Disponible: https://www.ni.com/docs/en-US/bundle/labview/page/lvconcepts/shift_registers_concepts.html

[9] Xilinx (2017, 20 de diciembre). Vivado Design Suite User Guide: Power Analysis and Optimization. [Online]. Disponible: https://www.xilinx.com/support.html#documentation

[10] Xilinx (2018, 6 de junio). Vivado Design Suite User Guide: Using Constraints.

[Online] Disponible: https://www.xilinx.com/support.html#documentation

Capítulo 8. Anexos


sw_ruido[3:0]
clk
fpga_rx
reset
UART_RX_unit
UART_RX
modulacion_reg[7:0]
RTL_REG_ASYNC
DivFreq_Trama_unit
DivFreq_Trama
modulador_unit
modulador
gen_datos_unit
gen_datos
gen_ruido_unit
gen_ruido
demodulador_unit
demodulador
Control_envio_unit
Control_envio
UART_TX_unit
UART_TX
fpga_tx

START

STOP

VISA resource name

COM3

Tasa de Baudios

115200

Error Out

status

code

1073676294

source

VISA Read in Entrenador_TFM.vi

Nº bytes recibidos

1

Nº bytes transmitidos

0

Número de paquetes (sin contar FF)

11

Modulacion

FSK

ENVIAR

Muestras totales

0

5000

Muestras a visualizar

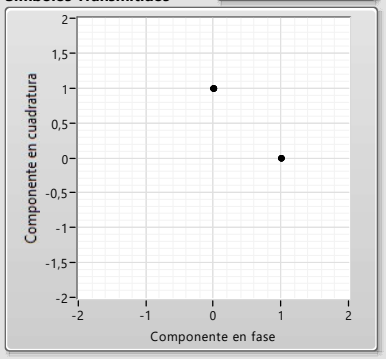
1000

Señal Modulada

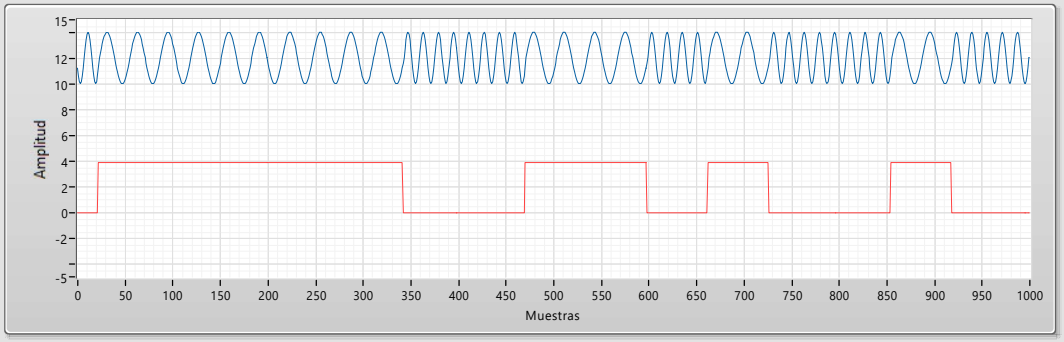
Datos transmitidos

Simbolos Transmitidos

Constelacion



Señales del Transmisor



SEÑAL MODULADA

Ganancia

Desplazamiento

DATOS TRANSMITIDOS

Ganancia

Desplazamiento

LIMPIAR GRAFICAS

Señal Recibida

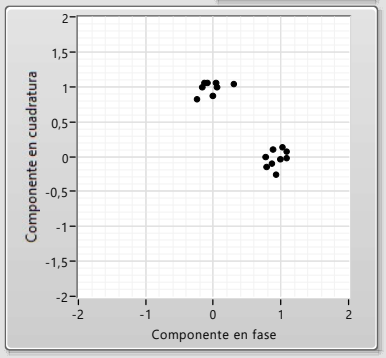
Datos demodulados

Canal X

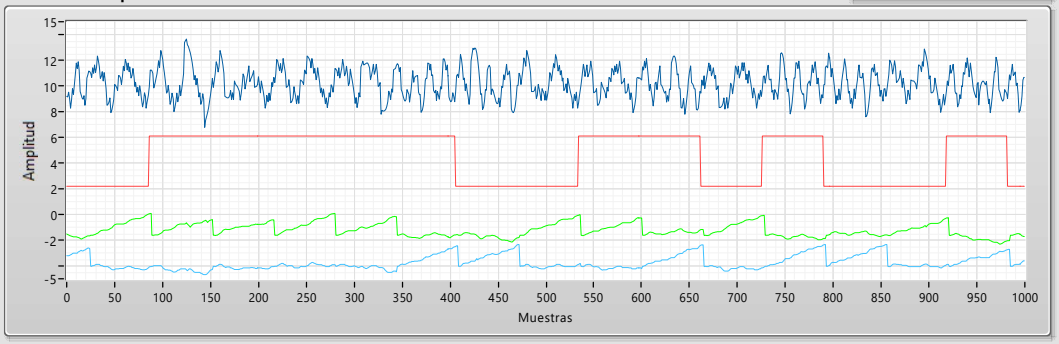
Canal Y

Simbolos Recibidos

Constelacion



Señales del Receptor



SEÑAL RECIBIDA

Ganancia

Desplazamiento

CANAL X

Ganancia

Desplazamiento

DATOS DEMODULADOS

Ganancia

Desplazamiento

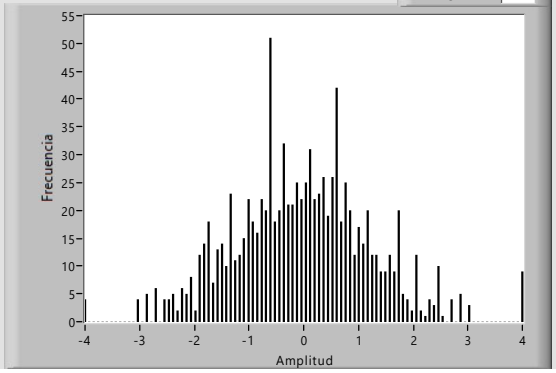
CANAL Y

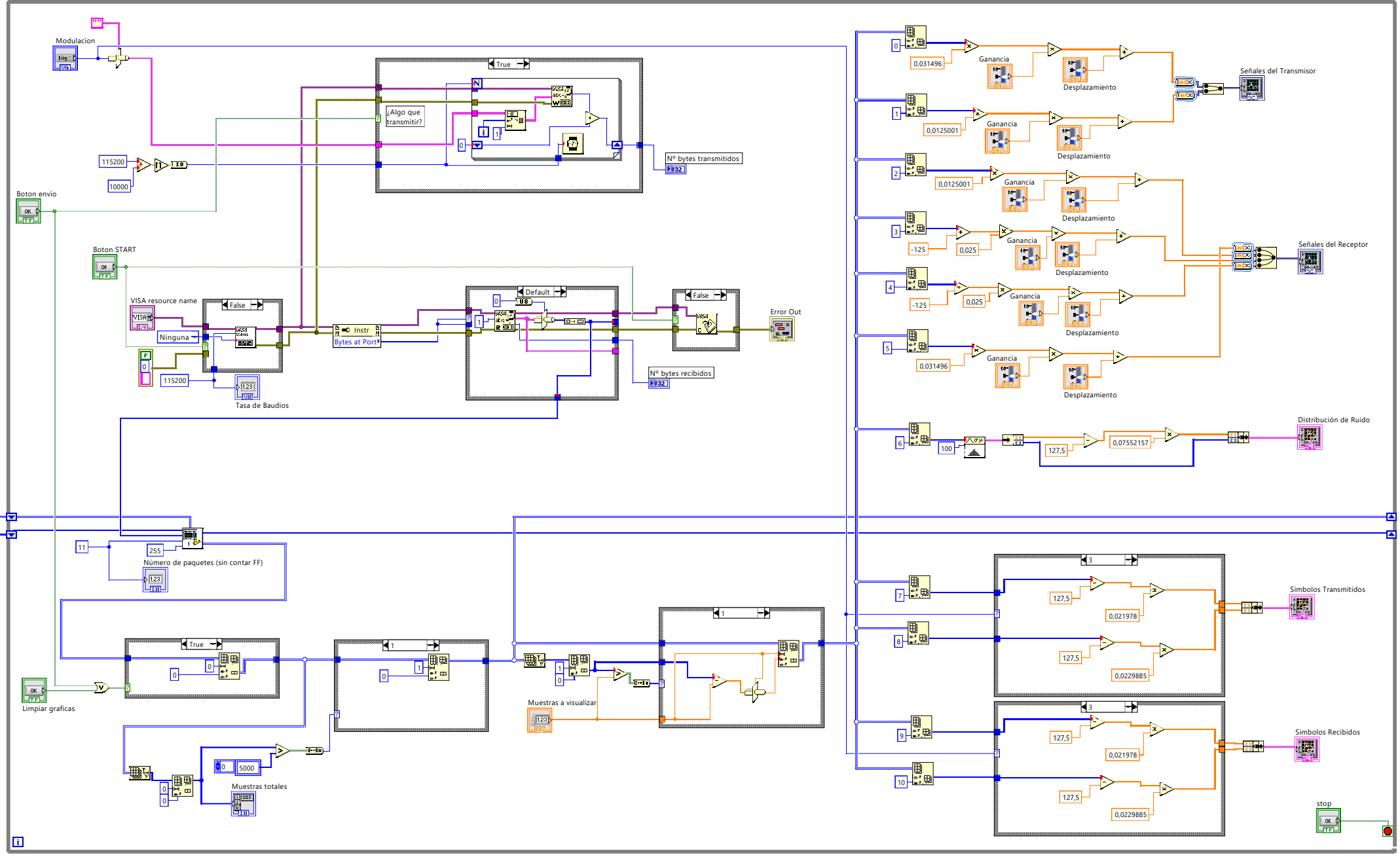
Ganancia

Desplazamiento

Distribución de Ruido

Histograma




Modulacion
True
¿Algo que transmitir?
115200
10000
Boton envio
Nº bytes transmitidos
Boton START
VISA resource name
False
Ninguna
115200
Tasa de Baudios
Instr
Bytes at Port
Default
U8
False
Error Out
Nº bytes recibidos
0,031496
Ganancia
Desplazamiento
Señales del Transmisor
0,0125001
-125
0,025
Señales del Receptor
100
127,5
0,07552157
Distribución de Ruido
11
255
Número de paquetes (sin contar FF)
Limpiar graficas
Muestras a visualizar
5000
Muestras totales
0,021978
0,0229885
Simbolos Transmitidos
Simbolos Recibidos
stop

```
N = 256;
x = linspace(-4,4,N);
pd = makedist('Normal');
p = cdf(pd,x);

pmin = min(p);
pmax = max(p);

q = 8;
D = 2^q;
f_s = floor((p-pmin)/(pmax-pmin+10*eps)*D);
f2 = (f_s+0.5)/D*(pmax-pmin)+pmin;
error = abs(f2-p);

plot(x,f_s,LineWidth=1.5)
axis([-4 4 0 255])
```

Módulo Top_Entrenador

```
-- Este modulo se corresponde al Top Level de todo el sistema.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_arith.ALL;
use WORK.constantes.ALL;
use WORK.real2bit.ALL;

entity Top_Entrenador is
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           sw_ruido : in STD_LOGIC_VECTOR(3 downto 0);
           fpga_tx : out STD_LOGIC;
           fpga_rx : in STD_LOGIC);
end Top_Entrenador;

architecture Behavioral of Top_Entrenador is

component DivFreq_Trama
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           clk_trama : out STD_LOGIC;
           clk_trama_x12 : out STD_LOGIC);
    end component;

component modulador
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           dato : in STD_LOGIC;
           dato_2bits : in STD_LOGIC_VECTOR(1 downto 0);
           modulacion : in std_logic_vector (7 downto 0);
           signal_modulada : out palabra;
           constelacion_x : out STD_LOGIC_VECTOR (7 downto 0);
           constelacion_y : out STD_LOGIC_VECTOR (7 downto 0);
           periodo : out STD_LOGIC);
    end component;
component gen_datos
    Port ( clk : in STD_LOGIC;
           clk_periodo : in STD_LOGIC;
           reset : in STD_LOGIC;
           dato : out STD_LOGIC;
           dato_2bits : out STD_LOGIC_VECTOR(1 downto 0);
           dato_8bits : out STD_LOGIC_VECTOR(7 downto 0));
    end component;

component gen_ruido
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           sw_ruido : in STD_LOGIC_VECTOR(3 downto 0);
           dato_8bits : in STD_LOGIC_VECTOR (7 downto 0);
           signal_modulada : in palabra;
           signal_ruido : out palabra;
           amplitud_ruido : out palabra);
    end component;

component Control_envio
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           modulacion : in STD_LOGIC_VECTOR(7 downto 0);
           dato : in STD_LOGIC;
           dato_2bits : in STD_LOGIC_VECTOR(1 downto 0);
           transmitiendo : in STD_LOGIC;
           signal_modulada : in palabra;
           signal_ruido : in palabra;
           ruido : in palabra;
           clk_trama : in std_logic;
           signal_dem : in STD_LOGIC_VECTOR(1 downto 0);
           canal_x : in signed (7 downto 0);
           canal_y : in signed (7 downto 0);
           constelacion_x : in STD_LOGIC_VECTOR (7 downto 0);
           constelacion_y : in STD_LOGIC_VECTOR (7 downto 0);
           constelacion_ruido_x : in STD_LOGIC_VECTOR (7 downto 0);
           constelacion_ruido_y : in STD_LOGIC_VECTOR (7 downto 0);
           transmite : out STD_LOGIC;
           datos_8bits : out STD_LOGIC_VECTOR (7 downto 0));
    end component;
component demodulador
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           modulacion : in std_logic_vector (7 downto 0);
           signal_ruido : in palabra;
           canal_x : out signed (7 downto 0);
           canal_y : out signed (7 downto 0);
           constelacion_ruido_x : out STD_LOGIC_VECTOR (7 downto 0);
           constelacion_ruido_y : out STD_LOGIC_VECTOR (7 downto 0);
           signal_dem : out STD_LOGIC_VECTOR(1 downto 0));
    end component;

component UART_TX
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           transmite : in STD_LOGIC;
           dato_tx_in : in STD_LOGIC_VECTOR (7 downto 0);
           transmitiendo : out STD_LOGIC;
           fpga_tx : out STD_LOGIC);
    end component;

component UART_RX
    Port (clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           fpga_rx : in STD_LOGIC;
           dato_rx_out : out STD_LOGIC_VECTOR (7 downto 0);
           aviso_rx : out STD_LOGIC;
           recibiendo : out STD_LOGIC);
    end component;

signal clk_trama, clk_trama_x12, dato, transmite, transmitiendo, clk_periodo, aviso_rx, recibiendo : std_logic;
signal datos_8bits, dato_8bits_random, dato_rx_out, constelacion_ruido_x, constelacion_ruido_y : std_logic_vector(7 downto 0);
signal modulacion : std_logic_vector(7 downto 0) := (others => '0');
signal canal_x, canal_y : signed(7 downto 0) := (others => '0');
signal signal_modulada, signal_ruido, ruido : palabra;
signal dato_2bits, signal_dem : std_logic_vector(1 downto 0);

signal constelacion_x, constelacion_y : STD_LOGIC_VECTOR(7 downto 0);

signal signal_modulada_unsigned : std_logic_vector(7 downto 0);
```

```
begin

-- Para transmitir los 12 paquetes, lo que se ha hecho ha sido aguantar el valor de cada paquete lo que durarian en enviarse 12 tramas.
-- Por tanto, el reloj del modulador, del generador de datos y de ruido y del demodulador, dependen de clk_trama_x12.

DivFreq_Trama_unit : DivFreq_Trama
port map (
   clk => clk,
   reset => reset,
   clk_trama => clk_trama,
   clk_trama_x12 => clk_trama_x12
);

modulador_unit : modulador
port map (
   clk => clk_trama_x12,
   reset => reset,
   dato => dato,
   dato_2bits => dato_2bits,
   modulacion => modulacion,
   signal_modulada => signal_modulada,
   constelacion_x => constelacion_x,
   constelacion_y => constelacion_y,
   periodo => clk_periodo
);

gen_datos_unit : gen_datos
port map (
   clk => clk_trama_x12,
   clk_periodo => clk_periodo,
   reset => reset,
   dato => dato,
   dato_2bits => dato_2bits,
   dato_8bits => dato_8bits_random
);
gen_ruido_unit : gen_ruido
port map (
   clk => clk_trama_x12,
   reset => reset,
   sw_ruido => sw_ruido,
   dato_8bits => dato_8bits_random,
   signal_modulada => signal_modulada,
   signal_ruido => signal_ruido,
   amplitud_ruido => ruido
);

Control_envio_unit : Control_envio
port map (
   clk => clk,
   reset => reset,
   modulacion => modulacion,
   dato => dato,
   dato_2bits => dato_2bits,
   transmitiendo => transmitiendo,
   signal_modulada => signal_modulada,
   signal_ruido => signal_ruido,
   ruido => ruido,
   clk_trama => clk_trama,
   signal_dem => signal_dem,
   canal_x => canal_x,
   canal_y => canal_y,
   constelacion_x => constelacion_x,
   constelacion_y => constelacion_y,
   constelacion_ruido_x => constelacion_ruido_x,
   constelacion_ruido_y => constelacion_ruido_y,
   transmite => transmite,
   datos_8bits => datos_8bits
);
demodulador_unit : demodulador
port map (
   clk => clk_trama_x12,
   reset => reset,
   modulacion => modulacion,
   signal_ruido => signal_ruido, --signal_ruido, --signal_modulada,
   canal_x => canal_x,
   canal_y => canal_y,
   constelacion_ruido_x => constelacion_ruido_x,
   constelacion_ruido_y => constelacion_ruido_y,
   signal_dem => signal_dem
);

UART_TX_unit : UART_TX
port map (
   clk => clk,
   reset => reset,
   transmite => transmite,
   dato_tx_in => datos_8bits,
   transmitiendo => transmitiendo,
   fpga_tx => fpga_tx
);

UART_RX_unit : UART_RX
port map (
   clk => clk,
   reset => reset,
   fpga_rx => fpga_rx,
   dato_rx_out => dato_rx_out,
   aviso_rx => aviso_rx,
   recibiendo => recibiendo
);
-- RECEPCION DEL TIPO DE MODULACION QUE HAYA SELECCIONADO EL CLIENTE

process(clk, reset)
begin

    if reset = '1' then
        modulacion <= (others => '0');
    elsif clk'event and clk='1' then
        if aviso_rx = '1' then
            modulacion <= dato_rx_out;
        end if;
    end if;

end process;

end Behavioral;
```

Módulo DivFreq_Trama

```
-- Este bloque se encarga de generar dos señales de reloj:
-- - clk_trama. Es el reloj cuyo periodo se corresponde a lo que tarda en enviarse una trama.
-- - clk_tramax12. Es el reloj cuyo periodo se corresponde a lo que tarda en enviarse 12 paquetes.

library IEEE, WORK;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_arith.ALL;
use WORK.UART_PKG.ALL;

entity DivFreq_Trama is
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           clk_trama : out STD_LOGIC;
           clk_trama_x12 : out STD_LOGIC
           );
end DivFreq_Trama;

architecture Behavioral of DivFreq_Trama is

signal cont_baud : unsigned (16 downto 0);
signal clk_aux : std_logic;

signal cont_baud_tramas : unsigned (3 downto 0);
signal clk_aux_tramas : std_logic;

begin

-- El valor de cont_baud se corresponde al valor de la cuenta que corresponde al tiempo que tarda en enviarse una trama.
-- La cuenta mínima necesaria aproximada para enviar una sola trama es (Reloj de la placa/Tasa de trama): 100MHz/11520 = 8680.
-- La tasa de trama se obtiene de dividir entre 10 la tasa de baudios (ya que se envían 10 bits en una trama).
-- Por tanto, se ha estableoido una cuenta mucho mayor para que no se introduzcan errores de envío (una cuenta de 50000)
P_DivFreq: Process(reset,clk)
begin
    if reset = '1' then
        cont_baud <= (others => '0');
        clk_aux <= '0';
    elsif clk'event and clk='1' then
            if cont_baud = 50000-1 then
                clk_aux <= not(clk_aux);
                cont_baud <= (others => '0');
            elsif cont_baud = (50000/2) - 1 then            -- Se niega clk_aux a mitad de valor de la cuenta para generar la señal de reloj.
                clk_aux <= not(clk_aux);
                cont_baud <= cont_baud + 1;
            else
                cont_baud <= cont_baud + 1;
            end if;
    end if;
end process;

clk_trama <= clk_aux;

-- El valor de cont_baud_tramas se corresponde al numero de paquetes - 1 que se quiere enviar.
-- NOTA: Este valor debe ser numero PAR ya que al dividirlo por dos no debe ser decimal.
P_DivFreq_Tramas: Process(reset,clk_aux)
begin
    if reset = '1' then
        cont_baud_tramas <= (others => '0');
        clk_aux_tramas <= '1';
    elsif clk_aux'event and clk_aux='1' then
            if cont_baud_tramas = 11 then
                clk_aux_tramas <= not(clk_aux_tramas);
                cont_baud_tramas <= (others => '0');
            elsif cont_baud_tramas = 5 then
                clk_aux_tramas <= not(clk_aux_tramas);
                cont_baud_tramas <= cont_baud_tramas + 1;
            else
                cont_baud_tramas <= cont_baud_tramas + 1;
            end if;
    end if;
end process;

clk_trama_x12 <= clk_aux_tramas;

end Behavioral;
```

Módulo modulador

```
-- Este bloque es el encargado de generar la señal modulada en funcion del tipo de modulacion que se haya escogido (recibido desde el modulo de recepcion UART)
-- De este bloque, las salidas a enviar son la señal modulada y las constelaciones sin ruido.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use WORK.constantes.ALL;
use WORK.real2bit.ALL;

entity modulador is
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           dato : in STD_LOGIC;
           dato_2bits : in STD_LOGIC_VECTOR(1 downto 0);
           modulacion : in std_logic_vector (7 downto 0);
           signal_modulada : out palabra;
           constelacion_x : out STD_LOGIC_VECTOR (7 downto 0);
           constelacion_y : out STD_LOGIC_VECTOR (7 downto 0);
           periodo : out STD_LOGIC);
end modulador;

architecture Behavioral of modulador is

    signal cont_periodo : natural;
    signal aux_periodo : std_logic;

    signal puntero : natural range 0 to (M-1) := 0;             -- Puntero para generar una señal senoidal
    signal puntero_espacio : natural range 0 to (M-1) := 0;     -- Puntero para generar una señal senoidal de doble frecuencia (FSK)
    signal valor : palabra := (others => '0');

    signal puntero_45 : natural range 0 to (M-1) := 0;          -- Puntero para generar una señal senoidal desfasada 45º
    signal puntero_135 : natural range 0 to (M-1) := 0;         -- Puntero para generar una señal senoidal desfasada 135º
    signal puntero_255 : natural range 0 to (M-1) := 0;         -- Puntero para generar una señal senoidal desfasada 255º
    signal puntero_315 : natural range 0 to (M-1) := 0;         -- Puntero para generar una señal senoidal desfasada 315º

begin
-- Este proceso genera un reloj que marca los ciclos que tiene la señal en un bit de dato.
-- Como el periodo es 64 y la tabla de generacion del seno tiene 32 posiciones, un bit de dato tendrá dos ciclos de la señal senoidal.
     process(clk, reset, cont_periodo)
     begin
        if(reset = '1') then
            cont_periodo <= 0;
            aux_periodo <= '1';
        elsif clk'event and clk = '1' then
            if cont_periodo = 63 then
                cont_periodo <= 0;
                aux_periodo <= not(aux_periodo);
            elsif cont_periodo = 31 then
                aux_periodo <= not(aux_periodo);
                cont_periodo <= cont_periodo + 1;
            else
                cont_periodo <= cont_periodo + 1;
            end if;
        end if;
     end process;

-- Modulador
     process(clk, reset, modulacion, dato, dato_2bits)
     begin
        if(reset = '1') then
            puntero <= 0;
            puntero_espacio <= 0;
            constelacion_x <= (others => '0');
            constelacion_y <= (others => '0');
            valor <= (others => '0');

        elsif clk'event and clk = '1' then

            puntero <= (puntero + 1) mod M;
            puntero_espacio <= (puntero_espacio + 2) mod M;

            puntero_45 <= (puntero + 5) mod M;
            puntero_135 <= (puntero + 13) mod M;
            puntero_255 <= (puntero + 21) mod M;
            puntero_315 <= (puntero + 29) mod M;

            -- Los valores de las constelaciones se han obtenido demodulando la señal sin ruido y viendo el valor final del integrador.

            if modulacion = "00000000" then -- MODULACION ASK.
                if dato = '1' then          -- Si el dato es '1', muestra el seno.
                    valor <= tabla_onda(puntero);
                    constelacion_x <= "10111111"; -- 191
                    constelacion_y <= "10000000"; -- 128
                else                        -- Si el dato es '0', muestra el valor medio de la señal. (En unsigned, el 0 es 128).
                    valor <= (others => '0'); --128
                    constelacion_x <= "10000000"; -- 128
                    constelacion_y <= "10000000"; -- 128
                end if;
            elsif modulacion = "00000001" then -- MODULACION BPSK
                if dato = '1' then          -- Si el dato es '1', muestra el seno.
                    valor <= tabla_onda(puntero);
                    constelacion_x <= "10111111"; -- 191
                    constelacion_y <= "10000000"; -- 128
                else                        -- Si el dato es '0', muestra el menos seno.
                    valor <= -tabla_onda(puntero);
                    constelacion_x <= "01000000"; -- 64
                    constelacion_y <= "10000000"; -- 128
                end if;
            elsif modulacion = "00000010" then -- MODULACION FSK
                if dato = '1' then          -- Si el dato es '1', muestra el seno.
                    valor <= tabla_onda(puntero);
                    constelacion_x <= "10111111"; -- 191
                    constelacion_y <= "01111111"; -- 127
                else                        -- Si el dato es '0', muestra el seno con el doble de frecuencia.
                    valor <= tabla_onda(puntero_espacio);
                    constelacion_x <= "01111111"; -- 127
                    constelacion_y <= "10111111"; -- 191
                end if;
```

```
            elsif modulacion = "00000011" then -- QPSK
            -- PARA ESTA MODULACION SE HA UTILIZADO LA CODIFICACION GRAY
                if dato_2bits = "00" then           -- Si el dato es '00', muestra el seno desfasado 45°.
                    valor <= tabla_onda(puntero_45);
                    constelacion_x <= "10101101"; -- 173
                    constelacion_y <= "10101011"; -- 171
                elsif dato_2bits = "01" then        -- Si el dato es '01', muestra el seno desfasado 135°.
                    valor <= tabla_onda(puntero_135);
                    constelacion_x <= "01010010"; -- 82
                    constelacion_y <= "10101011"; -- 171
                elsif dato_2bits = "11" then        -- Si el dato es '11', muestra el seno desfasado 255°.
                    valor <= tabla_onda(puntero_255);
                    constelacion_x <= "01010010"; -- 82
                    constelacion_y <= "01010100"; -- 84
                elsif dato_2bits = "10" then        -- Si el dato es '10', muestra el seno desfasado 315°.
                    valor <= tabla_onda(puntero_315);
                    constelacion_x <= "10101101"; -- 173
                    constelacion_y <= "01010100"; -- 84
                else
                    valor <= (others => '0');
                    constelacion_x <= (others => '0');
                    constelacion_y <= (others => '0');
                end if;
            else
                valor <= (others => '0');
                constelacion_x <= (others => '0');
                constelacion_y <= (others => '0');
            end if;
        end if;
    end process;

signal_modulada <= valor;
periodo <= aux_periodo;

end Behavioral;
```

Módulo gen_datos

```
-- Este modulo se encarga de generar los datos pseudoaleatorios tanto para la generacion del ruido
-- como para la modulacion de las señales.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.constantes.ALL;

entity gen_datos is
    generic (Nreg : positive := N;
             Nreg_8 : positive := N_8);
    Port ( clk : in STD_LOGIC;
           clk_periodo : in STD_LOGIC;
           reset : in STD_LOGIC;
           dato : out STD_LOGIC;                                -- La señal dato son datos pseudoaleatorios de 1 bit (ASK, BPSK, FSK)
           dato_2bits : out STD_LOGIC_VECTOR(1 downto 0);    -- La señal dato_2bits son datos pseudoaleatorios de 2 bits (QPSK)
           dato_8bits : out STD_LOGIC_VECTOR(7 downto 0)     -- La señal dato_8bits son datos pseudoaleatorios de 8 bits (Para el ruido)
           );
end gen_datos;

architecture Behavioral of gen_datos is

    component registro is
        port ( clk, preset, D : in STD_LOGIC;
                            Q : out STD_LOGIC);
    end component registro;

    signal sig_xor : std_logic;
    signal Q_int : std_logic_vector(Nreg-1 downto 0);

    signal sig_xor_8 : std_logic;
    signal Q_int_8 : std_logic_vector(Nreg_8-1 downto 0);

begin

-- Se hace un bucle for para no instanciar el componente N veces.
-- GENERADOR DE DATOS PARA LAS MODULACIONES
    Generador_datos : for I in 0 to Nreg - 1 generate
        Reg00 : if (I=0) generate
            Reg0 : Registro
            port map (
                clk => clk_periodo,
                preset => reset,
                D => sig_xor,
                Q => Q_int(0)
            );
        end generate;
        Regs : if I>0 generate
            Reg : Registro
            port map (
                clk => clk_periodo,
                preset => reset,
                D => Q_int(I-1),
                Q => Q_int(I)
            );
        end generate;
    end generate;

    dato <= Q_int(Nreg-1);
    dato_2bits <= Q_int(1 downto 0);
    sig_xor <= Q_int(1) xor Q_int(Nreg-1);
-- GENERADOR DE DATOS PARA EL RUIDO
    Generador_datos_8bits : for I in 0 to Nreg_8 - 1 generate
        Reg00 : if (I=0) generate
            Reg0 : Registro
            port map (
                clk => clk,
                preset => reset,
                D => sig_xor_8,
                Q => Q_int_8(0)
            );
        end generate;
        Regs : if I>0 generate
            Reg : Registro
            port map (
                clk => clk,
                preset => reset,
                D => Q_int_8(I-1),
                Q => Q_int_8(I)
            );
        end generate;
    end generate;

    dato_8bits <= Q_int_8(7 downto 0);
    sig_xor_8 <= Q_int_8(20) xor Q_int_8(18);

end Behavioral;
```

Módulo registro

```
-- Biestable tipo D

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity registro is
    Port ( clk : in STD_LOGIC;
           preset : in STD_LOGIC;
           D : in STD_LOGIC;
           Q : out STD_LOGIC);
end registro;

architecture Behavioral of registro is

begin
    process(clk, preset)
    begin
        if preset = '1' then
            Q <= '1';
        elsif clk'event and clk = '1' then
            Q <= D;
        end if;
    end process;
end Behavioral;
```

Módulo gen_ruido

```vhdl
-- Este modulo se encarga de generar la amplitud de ruido y de añadir ese ruido a la señal modulada.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_arith.ALL;
use IEEE.std_logic_signed.ALL;
use WORK.constantes.ALL;
use WORK.real2bit.ALL;

entity gen_ruido is
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           sw_ruido : in STD_LOGIC_VECTOR(3 downto 0);
           dato_8bits : in STD_LOGIC_VECTOR (7 downto 0);
           signal_modulada : in palabra;
           signal_ruido : out palabra;
           amplitud_ruido : out palabra
           );
end gen_ruido;

architecture Behavioral of gen_ruido is

    signal ruido : palabra := (others => '0');

begin

process(clk, reset)
begin
    if(reset = '1') then
        ruido <= (others => '0');
    elsif clk'event and clk = '1' then
        ruido <= tabla_distribucion(conv_integer(unsigned(dato_8bits)));     -- La amplitud de ruido se obtiene de la tabla de la distribucion normal de la libreria real2bit
    end if;                                                                  -- se utiliza conv_integer(unsigned(dato_8bits)) para convertirlo en integer
end process;

process(sw_ruido, signal_modulada, ruido)
begin
    if sw_ruido = "0000" then
        signal_ruido <= signal_modulada;
        amplitud_ruido <= (others => '0');
    elsif sw_ruido = "0001" then
        signal_ruido <= signal_modulada + extraer(ruido*truncar(0.066667*1));
        amplitud_ruido <= extraer(ruido*truncar(0.066667*1));
    elsif sw_ruido = "0010" then
        signal_ruido <= signal_modulada + extraer(ruido*truncar(0.066667*2));
        amplitud_ruido <= extraer(ruido*truncar(0.066667*2));
    elsif sw_ruido = "0011" then
        signal_ruido <= signal_modulada + extraer(ruido*truncar(0.066667*3));
        amplitud_ruido <= extraer(ruido*truncar(0.066667*3));
    elsif sw_ruido = "0100" then
        signal_ruido <= signal_modulada + extraer(ruido*truncar(0.066667*4));
        amplitud_ruido <= extraer(ruido*truncar(0.066667*4));
    elsif sw_ruido = "0101" then
        signal_ruido <= signal_modulada + extraer(ruido*truncar(0.066667*5));
        amplitud_ruido <= extraer(ruido*truncar(0.066667*5));
    elsif sw_ruido = "0110" then
        signal_ruido <= signal_modulada + extraer(ruido*truncar(0.066667*6));
        amplitud_ruido <= extraer(ruido*truncar(0.066667*6));
    elsif sw_ruido = "0111" then
        signal_ruido <= signal_modulada + extraer(ruido*truncar(0.066667*7));
        amplitud_ruido <= extraer(ruido*truncar(0.066667*7));
    elsif sw_ruido = "1000" then
        signal_ruido <= signal_modulada + extraer(ruido*truncar(0.066667*8));
        amplitud_ruido <= extraer(ruido*truncar(0.066667*8));
    elsif sw_ruido = "1001" then
        signal_ruido <= signal_modulada + extraer(ruido*truncar(0.066667*9));
        amplitud_ruido <= extraer(ruido*truncar(0.066667*9));
    elsif sw_ruido = "1010" then
        signal_ruido <= signal_modulada + extraer(ruido*truncar(0.066667*10));
        amplitud_ruido <= extraer(ruido*truncar(0.066667*10));
    elsif sw_ruido = "1011" then
        signal_ruido <= signal_modulada + extraer(ruido*truncar(0.066667*11));
        amplitud_ruido <= extraer(ruido*truncar(0.066667*11));
    elsif sw_ruido = "1100" then
        signal_ruido <= signal_modulada + extraer(ruido*truncar(0.066667*12));
        amplitud_ruido <= extraer(ruido*truncar(0.066667*12));
    elsif sw_ruido = "1101" then
        signal_ruido <= signal_modulada + extraer(ruido*truncar(0.066667*13));
        amplitud_ruido <= extraer(ruido*truncar(0.066667*13));
    elsif sw_ruido = "1110" then
        signal_ruido <= signal_modulada + extraer(ruido*truncar(0.066667*14));
        amplitud_ruido <= extraer(ruido*truncar(0.066667*14));
    elsif sw_ruido = "1111" then
        signal_ruido <= signal_modulada + ruido;
        amplitud_ruido <= ruido;
    else
        signal_ruido <= signal_modulada;
        amplitud_ruido <= (others => '0');
    end if;
end process;

end Behavioral;
```

Módulo control_envio

```
-- Este modulo se encarga de organizar el orden de envio de los paquetes a la UART.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_arith.ALL;
use IEEE.std_logic_signed.ALL;
use WORK.constantes.ALL;
use WORK.real2bit.ALL;

entity Control_envio is
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           modulacion : in STD_LOGIC_VECTOR(7 downto 0);
           dato : in STD_LOGIC;
           dato_2bits : in STD_LOGIC_VECTOR(1 downto 0);
           transmitiendo : in STD_LOGIC;
           signal_modulada : in palabra;
           signal_ruido : in palabra;
           ruido : in palabra;
           clk_trama : in std_logic;
           signal_dem : in STD_LOGIC_VECTOR(1 downto 0);
           canal_x : in signed (7 downto 0);
           canal_y : in signed (7 downto 0);
           constelacion_x : in STD_LOGIC_VECTOR (7 downto 0);
           constelacion_y : in STD_LOGIC_VECTOR (7 downto 0);
           constelacion_ruido_x : in STD_LOGIC_VECTOR (7 downto 0);
           constelacion_ruido_y : in STD_LOGIC_VECTOR (7 downto 0);
           transmite : out STD_LOGIC;
           datos_8bits : out STD_LOGIC_VECTOR (7 downto 0));
end Control_envio;

architecture Behavioral of Control_envio is

signal cont : unsigned(3 downto 0);
signal signal_modulada_unsigned : std_logic_vector(7 downto 0);
signal canal_x_unsigned : std_logic_vector(7 downto 0);
signal canal_y_unsigned : std_logic_vector(7 downto 0);
signal signal_ruido_unsigned : std_logic_vector(7 downto 0);
signal ruido_unsigned : std_logic_vector(7 downto 0);

signal B, Z, FlancoUp : std_logic;
begin

-- Este proceso se encarga de contar los paquetes que se van enviando en cada trama
process (clk_trama, reset)
begin
    if reset = '1' then
        cont <= (others => '0');
    elsif clk_trama'event and clk_trama='1' then
        if cont = 11 then
            cont <= (others => '0');
        else
            cont <= cont + 1;
        end if;
    end if;
end process;

-- Como solo se envian 8 bits por paquete, se escogen los 8 bits MSB y se convierten en signed (Ca2) con el comando conv_signed.
signal_modulada_unsigned <= signal_modulada(11 downto 4) + conv_signed(2**(8-1),8);
canal_x_unsigned <= canal_x + conv_signed(2**(8-1),8);
canal_y_unsigned <= canal_y + conv_signed(2**(8-1),8);
signal_ruido_unsigned <= signal_ruido(11 downto 4) + conv_signed(2**(8-1),8);
ruido_unsigned <= ruido(11 downto 4) + conv_signed(2**(8-1),8);

-------------------- DETECTOR DE FLANCO --------------------
-- Se introduce un detector de flanco de manera que la señal 'transmite' solo dure un ciclo de reloj para el buen funcionamiento
-- del modulo de transmision de la UART.

P_BIEST: process (reset, clk)
begin
    if reset = '1' then
        B <= '0';
        Z <= '0';
    elsif clk'event and clk = '1' then
        B <= clk_trama;
        Z <= B;
    end if;
end process;

FlancoUp <= '1' when Z = '1' and B = '0' else '0';

process(FlancoUp, transmitiendo)
begin

    if FlancoUp = '1' then
        transmite <= '1';
    elsif transmitiendo = '1' then
        transmite <= '0';
    end if;

end process;

-------------------- CONTROL DEL ENVIO DE LOS PAQUETES DE DATOS --------------------
-- Como requisitio de comunicacion con el programa LabView, se debe enviar un bit de inicio que marque el principio de los paquetes cuyo valor sea FF.
process (clk, reset)
begin
    if reset = '1' then
        datos_8bits <= (others => '0');
    elsif clk'event and clk='1' then
        if transmitiendo = '0' then
            if cont = 0 then                        -- BYTE DE INICIO (FF)
                datos_8bits <= "11111111"; --255

            elsif cont = 1 then                     -- SEÑAL MODULADA
                datos_8bits <= signal_modulada_unsigned;
```

```
            elsif cont = 2 then                     -- SEÑAL DE DATOS
                if modulacion = "00000011" then -- QPSK (2 bits)
                    if dato_2bits = "11" then
                        datos_8bits <= "11110000"; -- 240
                    elsif dato_2bits = "10" then
                        datos_8bits <= "10100000"; -- 160
                    elsif dato_2bits = "01" then
                        datos_8bits <= "01010000"; -- 80
                    else
                        datos_8bits <= (others => '0');
                    end if;
                else                            -- Resto de modulaciones (1 bit)
                    if dato = '1' then
                        datos_8bits <= "11110000"; -- 240
                    else
                        datos_8bits <= (others => '0');
                    end if;
                end if;

            elsif cont = 3 then                     -- SEÑAL DEMODULADA
                if signal_dem = "11" then
                    datos_8bits <= "11110000"; -- 240
                elsif signal_dem = "10" then
                    datos_8bits <= "10100000"; -- 160
                elsif signal_dem = "01" then
                    datos_8bits <= "01010000"; -- 80
                else
                    datos_8bits <= (others => '0'); -- 0
                end if;

            elsif cont = 4 then                     -- CANAL X
                datos_8bits <= canal_x_unsigned;

            elsif cont = 5 then                     -- CANAL Y
                datos_8bits <= canal_y_unsigned;

            elsif cont = 6 then                     -- SEÑAL MODULADA CON RUIDO
                datos_8bits <= signal_ruido_unsigned;

            elsif cont = 7 then                     -- SEÑAL DE RUIDO
                datos_8bits <= ruido_unsigned;

            elsif cont = 8 then                     -- CONSTELACION X SIN RUIDO
                datos_8bits <= constelacion_x;

            elsif cont = 9 then                     -- CONSTELACION Y SIN RUIDO
                datos_8bits <= constelacion_y;

            elsif cont = 10 then                     -- CONSTELACION X CON RUIDO
                datos_8bits <= constelacion_ruido_x;

            elsif cont = 11 then                     -- CONSTELACION Y CON RUIDO
                datos_8bits <= constelacion_ruido_y;
            end if;
        end if;
    end if;

end process;

end Behavioral;
```

Módulo demodulador

```
-- Este bloque se encarga de demodular la señal modulada.
-- De este bloque, las salidas a enviar son el canal x, el canal y, las constelaciones con ruido y la señal demodulada.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_arith.ALL;
use IEEE.std_logic_signed.ALL;
use WORK.constantes.ALL;
use WORK.real2bit.ALL;

entity demodulador is
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           modulacion : in std_logic_vector (7 downto 0);
           signal_ruido : in palabra;
           canal_x : out signed (7 downto 0);
           canal_y : out signed (7 downto 0);
           constelacion_ruido_x : out STD_LOGIC_VECTOR (7 downto 0) := (others => '0');
           constelacion_ruido_y : out STD_LOGIC_VECTOR (7 downto 0) := (others => '0');
           signal_dem : out STD_LOGIC_VECTOR(1 downto 0));
end demodulador;

architecture Behavioral of demodulador is

    signal producto_x : palabra := (others => '0');
    signal producto_y : palabra := (others => '0');

    signal puntero : natural range 0 to (M-1) := 0;
    signal portadora_canal_x : palabra := (others => '0');
    signal puntero_espacio : natural range 0 to (M-1) := 0;
    signal puntero_coseno : natural range 0 to (M-1) := 0;
    signal portadora_canal_y : palabra := (others => '0');

    signal cont_x : signed((2*nbits)-1 downto 0) := (others => '0');
    signal cont_y : signed ((2*nbits)-1 downto 0) := (others => '0');
    signal cont_periodo, cont_periodo_aux, cont_periodo_producto, cont_periodo_int : natural;
    signal error : std_logic;   -- Esta señal solo se usa para detectar errores

    signal salida_integrador_x : signed((2*nbits)-1 downto 0) := (others => '0');
    signal salida_integrador_y : signed((2*nbits)-1 downto 0) := (others => '0');
begin

-- Se obtiene la portadora de los canales
process(clk, reset, modulacion)
begin
    if(reset = '1') then
        portadora_canal_x <= (others => '0');
        portadora_canal_y <= (others => '0');
    elsif clk'event and clk = '1' then
        puntero <= (puntero + 1) mod M;                 -- SENO
        puntero_espacio <= (puntero_espacio + 2) mod M; -- SENO DOBLE FREQ
        puntero_coseno <= (puntero + 9) mod M;          -- COSENO.

        if modulacion = "00000010" then     -- FSK
            portadora_canal_x <= tabla_onda(puntero);
            portadora_canal_y <= tabla_onda(puntero_espacio);
        else                                -- ASK, BPSK (Solo se utiliza el canal x) y QPSK
            portadora_canal_x <= tabla_onda(puntero);
            portadora_canal_y <= tabla_onda(puntero_coseno);
        end if;
    end if;
end process;

-- Estos procesos se encargan de solventar el problema de los retrasos de las operaciones (Producto y suma)

process(reset, clk, cont_periodo)
begin
    if(reset = '1') then
        cont_periodo <= 0;
    elsif clk'event and clk = '1' then
        if cont_periodo = 63 then
           cont_periodo <= 0;
        else
            cont_periodo <= cont_periodo + 1;
        end if;
    end if;
end process;

process(reset, clk)
begin
    if reset = '1' then
        cont_periodo_aux <= 0;
        cont_periodo_producto <= 0;
        cont_periodo_int <= 0;
    elsif clk'event and clk = '1' then
        cont_periodo_aux <= cont_periodo;
        cont_periodo_producto <= cont_periodo_aux;
        cont_periodo_int <= cont_periodo_producto;
    end if;
end process;

-- Se multiplican la señal modulada y la portadora segun el canal
process(clk, reset, cont_periodo_producto)
begin
    if(reset = '1') then
        producto_x <= (others => '0');
        producto_y <= (others => '0');

    elsif clk'event and clk = '1' then
        if cont_periodo_producto = 63 then
            producto_x <= (others => '0');
            producto_y <= (others => '0');
        else
            producto_x <= extraer(signal_ruido * portadora_canal_x);
            producto_y <= extraer(signal_ruido * portadora_canal_y);
        end if;
    end if;
end process;
```

```
-- En el integrador, voy sumando las muestras. He puesto un contador de 64 cuentas porque es lo que dura un tiempo de bit.
process(reset, clk, modulacion, cont_periodo_int)
     begin
        if(reset = '1') then
            cont_x <= (others => '0');
            cont_y <= (others => '0');
        elsif clk'event and clk = '1' then
                if cont_periodo_int = 63 then
                    cont_x <= (others => '0');
                    cont_y <= (others => '0');
                elsif (modulacion = "00000010") or (modulacion = "00000011") then -- FSK y QPSK
                    cont_x <= cont_x + producto_x;
                    cont_y <= cont_y + producto_y;
                else                                                              -- ASK Y BPSK (no se usa el canal y)
                    cont_x <= cont_x + producto_x;
                    cont_y <= (others => '0');
                end if;
        end if;
end process;

-- El decisor decide si la señal demodulada es un 1 o un 0 al final de la cuenta.
-- En el resto de la cuenta aguanta el valor de la señal demodulada.
process(clk, reset, cont_periodo_int, cont_periodo, cont_x, cont_y,  modulacion) -- DECISOR
     begin
        if(reset = '1') then
            signal_dem <= (others => '0');
            error <= '0';
            salida_integrador_x <= (others => '0');
            salida_integrador_y <= (others => '0');

        elsif clk'event and clk = '1' then

            if modulacion = "00000000" then -- ASK
                if (cont_periodo_int = 63) then
                    salida_integrador_x <= cont_x;
                    salida_integrador_y <= cont_y;
                end if;
                if (cont_periodo = 63) then
                    if cont_x > 16336 then
                        signal_dem <= (others => '1');
                    else
                        signal_dem <= (others => '0');
                    end if;
                end if;

            elsif modulacion = "00000001" then -- BPSK
                if (cont_periodo_int = 63) then
                    salida_integrador_x <= cont_x;
                    salida_integrador_y <= cont_y;
                end if;
                if (cont_periodo = 63) then
                    if cont_x > 0 then
                        signal_dem <= (others => '1');
                    else
                        signal_dem <= (others => '0');
                    end if;
                end if;

            elsif modulacion = "00000010" then -- FSK
                if (cont_periodo_int = 63) then
                    salida_integrador_x <= cont_x;
                    salida_integrador_y <= cont_y;
                end if;
                if (cont_periodo = 63) then
                    if cont_x > 16536 then
                        signal_dem <= (others => '1');
                        error <= '0';
                    elsif cont_y > 15678 then
                        signal_dem <= (others => '0');
                        error <= '0';
                    else
                        signal_dem <= (others => '0');
                        error <= '1';
                    end if;
                end if;

            elsif modulacion = "00000011" then -- QPSK
                if (cont_periodo_int = 63) then
                    salida_integrador_x <= cont_x;
                    salida_integrador_y <= cont_y;
                end if;
                if (cont_periodo = 63) then
                    if ((cont_x >= 0) and (cont_y >= 0)) then
                        signal_dem <= "00";
                        error <= '0';
                    elsif (cont_x < 0) and (cont_y >= 0) then
                        signal_dem <= "01";
                        error <= '0';
                    elsif ((cont_x < 0) and (cont_y < 0)) then
                        signal_dem <= "11";
                        error <= '0';
                    elsif (cont_x >= 0) and (cont_y < 0) then
                        signal_dem <= "10";
                        error <= '0';
                    else
                        error <= '1';
                        signal_dem <= "11";
                    end if;
                end if;
            end if;
       end if;
end process;

-- Para los canales, solo se cogen 8 MSB de la cuenta para enviarlos por la UART como paquete.
canal_x <= cont_x(16 downto 9);
canal_y <= cont_y(16 downto 9);

-- Se guarda el ultimo valor del integrador para crear las constelaciones.
constelacion_ruido_x <= salida_integrador_x(16 downto 9) + conv_signed(2**(8-1),8);
constelacion_ruido_y <= salida_integrador_y(16 downto 9) + conv_signed(2**(8-1),8);

end Behavioral;
```

Módulo UART_TX

```
-- Este bloque se corresponde al Top level del transmisor del modulo UART

library IEEE, WORK;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_arith.ALL;
use WORK.UART_PKG.ALL;

entity UART_TX is
    Port ( clk : in STD_LOGIC;  -- Señal de reloj
           reset : in STD_LOGIC; -- Señal de reset asincrono
           transmite : in STD_LOGIC;     -- Señal del sistema que ordena al modulo la tarnsmision del dato que se encuentra en dato_tx_in. La orden sera de un unico ciclo de reloj.
           dato_tx_in : in STD_LOGIC_VECTOR (7 downto 0);    -- Dato que se quiere enviar, se proporciona cuando se activa transmite
           transmitiendo : out STD_LOGIC;    -- Indica al sistema que el modulo esta transmitiendo y por tanto no podra atender a ninguna nueva orden de transmision.
           fpga_tx : out STD_LOGIC);     -- Trama que se envia en serie al ordenador.
end UART_TX;

architecture Behavioral of UART_TX is

component DivFreq
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           en_divfreq : in STD_LOGIC;
           baud : out STD_LOGIC);
    end component;

component Control
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           baud : in STD_LOGIC;
           transmite : in STD_LOGIC;
           fin_cont8bits : in STD_LOGIC;
           en_divfreq : out STD_LOGIC;
           cargadato : out STD_LOGIC;
           desplaza : out STD_LOGIC;
           transmitiendo : out STD_LOGIC;
           estado_tx : out STD_LOGIC_VECTOR (1 downto 0));
    end component;

component CuentaBits
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           baud : in STD_LOGIC;
           estado_tx : in STD_LOGIC_VECTOR (1 downto 0);
           fin_cont8bits : out STD_LOGIC);
    end component;

component Carga_desplz
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           dato_tx_in : in STD_LOGIC_VECTOR (7 downto 0);
           cargadato : in STD_LOGIC;
           desplaza : in STD_LOGIC;
           dato_tx : out STD_LOGIC);
    end component;

component Seleccion
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           estado_tx : in STD_LOGIC_VECTOR (1 downto 0);
           dato_tx : in STD_LOGIC;
           fpga_tx : out STD_LOGIC);
    end component;

signal en_divfreq, baud, fin_cont8bits, cargadato, desplaza, dato_tx : std_logic;
signal estado_tx : std_logic_vector(1 downto 0);

begin

DivFreq_unit : DivFreq
port map (
   clk => clk,
   reset => reset,
   en_divfreq => en_divfreq,
   baud => baud
);
Control_unit : Control
port map (
   clk => clk,
   reset => reset,
   baud => baud,
   transmite => transmite,
   fin_cont8bits => fin_cont8bits,
   en_divfreq => en_divfreq,
   cargadato => cargadato,
   desplaza => desplaza,
   transmitiendo => transmitiendo,
   estado_tx => estado_tx
);

CuentaBits_unit : CuentaBits
port map (
   clk => clk,
   reset => reset,
   baud => baud,
   estado_tx => estado_tx,
   fin_cont8bits => fin_cont8bits
);

Carga_desplz_unit : Carga_desplz
port map (
   clk => clk,
   reset => reset,
   dato_tx_in => dato_tx_in,
   cargadato => cargadato,
   desplaza => desplaza,
   dato_tx => dato_tx
);
Seleccion_unit : Seleccion
port map (
   clk => clk,
   reset => reset,
   estado_tx => estado_tx,
   dato_tx => dato_tx,
   fpga_tx => fpga_tx
);

end Behavioral;
```

Módulo DivFreq

```
-- Este bloque genera la señal ('baud') que indica cuándo ha pasado el intervalo de tiempo correspondiente a cada bit de la trama de envio.
-- Esta señal tendra una frecuencia determinada por la constante c_baud y que se corresponde con los baudios a los que se
-- comunica la UART.

library IEEE, WORK;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_arith.ALL;
use WORK.UART_PKG.ALL;

entity DivFreq is
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           en_divfreq : in STD_LOGIC;
           baud : out STD_LOGIC);
end DivFreq;

-- Para diseñar el divisor de frecuencia se divide la frecuencia de entrada entre la frecuencia de salida.
-- 100MHz/115200 = 868,0555 -> 868. Este valor corresponde a la cuenta necesaria para obtener la frecuencia de salida.
-- 100MHz/868 = 115207,37Hz -> Periodo = 8,68us.
-- El final de la cuenta sería 867 (868-1 = 867).

architecture Behavioral of DivFreq is

signal cont_baud : unsigned (c_nb_cont_baud downto 0);

begin

P_DivFreq: Process(reset,clk)
begin
    if reset = '1' then -- Se ponen a cero las señales cuando el reset esté activo.
        cont_baud <= (others => '0');
        baud <= '0';
    elsif clk'event and clk='1' then
        if en_divfreq = '1' then     -- Solo hace la cuenta cuando la señal enable esté activa
            baud <= '0';
            if cont_baud = c_fin_cont_baud then -- Cuando llega al final de la cuenta, se reinicia, y 'baud' se pone a 1 (que indica el tiempo de envio de un bit)
                baud <= '1';
                cont_baud <= (others => '0');
            else
                cont_baud <= cont_baud + 1;
            end if;
        else
            cont_baud <= (others => '0');
        end if;
    end if;
end process;

end Behavioral;
```

Módulo CuentaBits

```
-- Este bloque se encarga de contar el numero de bit de datos que se transmiten, de manera que, cuando llega
-- al final de la cuenta, se indique mediante la señal fin_cont8bits.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_arith.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity CuentaBits is
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           baud : in STD_LOGIC;
           estado_tx : in STD_LOGIC_VECTOR (1 downto 0);
           fin_cont8bits : out STD_LOGIC);
end CuentaBits;

architecture Behavioral of CuentaBits is

signal cont : unsigned (2 downto 0);

begin

process(clk, reset, baud, estado_tx, cont)
begin

    if reset = '1' then
--        fin_cont8bits <= '0';
        cont <= (others => '0');
    elsif clk'event and clk='1' then
--        fin_cont8bits <= '0';
        if estado_tx = "10" and baud = '1' then -- La cuenta empieza a contar solo si se encuentra en el estado correspondiente
            if cont = 7 then                    -- y se haya terminado de enviar el primer bit de inicio.
                cont <= (others => '0');
--                fin_cont8bits <= baud;
            else
                cont <= cont + 1;
            end if;
        end if;
    end if;
end process;

fin_cont8bits <= '1' when (cont = 7 and baud ='1') else '0';     -- Lo puse asi para que se sincronice con la señal baud.
                                                                 -- Si se pone dentro del process, la señal se activa despues de que la señal baud se active.
end Behavioral;
```

Módulo Control

```
-- Este modulo se encarga de dirigir al resto de bloques. Es el que contiene la máquina de estados, de tipo Mealy,
-- que determina el funcionamiento del circuito de transmisión del modulo UART.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Control is
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           baud : in STD_LOGIC;
           transmite : in STD_LOGIC;
           transmitiendo : out STD_LOGIC;
           fin_cont8bits : in STD_LOGIC;
           en_divfreq : out STD_LOGIC;
           cargadato : out STD_LOGIC;
           desplaza : out STD_LOGIC;
           estado_tx : out STD_LOGIC_VECTOR (1 downto 0));
end Control;

architecture Behavioral of Control is

-- Se definen los cuatro estados:
-- · e_init. El transmisor se enouentra esperando la orden de transmisitr con la linea de transmision a uno.
-- · e_bit_init. Se envia el bit de inicio poniendo la linea a cero.
-- · e_bits_dato. Se envian los 8 bits de datos.
-- · e_bit_fin. Se envia el bit de fin poniendo la linea de transmision a uno.
type estados_TX is (e_init, e_bit_init, e_bits_dato, e_bit_fin);
signal estado_actual, estado_siguiente : estados_TX;

begin
-- MAQUINA DE ESTADOS TIPO MEALY (PORQUE LAS SALIDAS DEPENDEN DE LAS ENTRADAS)

-- En este proceso se obtiene el estado siguiente a partir del estado actual y las entradas.
-- Es un proceso combinacional, por tanto, no depende de la señal de reloj ni de reset.
P_COMB_ESTADO : process(estado_actual, transmite, baud, fin_cont8bits)
begin
    case estado_actual is
        when e_init =>
            if transmite = '1' then             -- La señal transmite marca el inicio de la transmision. Dura un unico ciclo de reloj.
                estado_siguiente <= e_bit_init; -- Una vez pasa de estado, transmite se pone a 0 y transmitiendo se pone a 1. Ignora a transmite si transmitiendo esta a 1.
            else
                estado_siguiente <= e_init;
            end if;
        when e_bit_init =>
            if baud = '1' then
                estado_siguiente <= e_bits_dato;
            else
                estado_siguiente <= e_bit_init;
            end if;
        when e_bits_dato =>
            if fin_cont8bits = '1' then
                estado_siguiente <= e_bit_fin;
            else
                estado_siguiente <= e_bits_dato;
            end if;
        when e_bit_fin =>
            if baud = '1' then
                estado_siguiente <= e_init;
            else
                estado_siguiente <= e_bit_fin;
            end if;
    end case;
end process;
-- Este proceso se encarga de actualizar el estado en cada ciclo de reloj y lo guarda en uno o varios
-- elementos de memoria. En cada ciclo de reloj, el estado siguiente pasa a ser el estado actual.
-- Como depende del reloj y de la señal de reset, es un proceso secuencial.
P_SEQ_FSM : process(clk, reset)
begin
    if reset = '1' then
        estado_actual <= e_init;
    elsif clk'event and clk = '1' then
        estado_actual <= estado_siguiente;
    end if;
end process;

-- Este proceso combinacional proporciona las salidas en funcion del estado actual y de las entradas, propio
-- de una máquina de estados tipo Mealy.
P_COM_SALIDAS : process(estado_actual, transmite, fin_cont8bits, baud)
begin
    case estado_actual is
        when e_init =>
            estado_tx <= "00";
            desplaza <= '0';
            en_divfreq <= '0';
            transmitiendo <= '0';
            if transmite = '1' then
                cargadato <= '1';
            else
                cargadato <= '0';
            end if;
        when e_bit_init =>
            estado_tx <= "01";
            cargadato <= '0';
            desplaza <= '0';
            en_divfreq <= '1';
            transmitiendo <= '1';
        when e_bits_dato =>
            estado_tx <= "10";
            cargadato <= '0';
            en_divfreq <= '1';
            transmitiendo <= '1';
            if fin_cont8bits = '0' then
                desplaza <= baud;
            else
                desplaza <= '0';
            end if;
        when e_bit_fin =>
            estado_tx <= "11";
            cargadato <= '0';
            desplaza <= '0';
            en_divfreq <= '1';
            transmitiendo <= '1';
    end case;
end process;

end Behavioral;
```

Módulo Carga_desplz

```
-- Este bloque se corresponde a un registro de desplazamiento de 8 bits hacia la izquierda, cuya orden de desplazamiento
-- lo marca la señal 'desplaza' y la orden de mandar el dato a la linea de transmision (dato_tx) lo da cargadato.
-- Al desplazar a la izquierda, lo que haya en el bit MSB se perderá, y lo que haya en el bit LSB se mantendrá, ya que
-- no se le asignará ningún nuevo valor.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Carga_desplz is
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           dato_tx_in : in STD_LOGIC_VECTOR (7 downto 0);
           cargadato : in STD_LOGIC;
           desplaza : in STD_LOGIC;
           dato_tx : out STD_LOGIC);
end Carga_desplz;

architecture Behavioral of Carga_desplz is

signal dato_tx_reg : std_logic_vector (7 downto 0) := (others =>'0');

begin

P_registro : Process (clk, reset)
begin
    if reset = '1' then
        dato_tx_reg <= (others => '0');
    elsif clk'event and clk = '1' then
        if cargadato = '1' then -- Al recibir la orden del bloque de control, se envía el dato a la linea de transmision
            dato_tx_reg <= dato_tx_in;
        elsif desplaza = '1' then -- Al recibir la orden del bloque de control, se desplaza hacia la izquierda cada vez que
                                  -- se reciba un '1' en esta señal.
            dato_tx_reg(6) <= dato_tx_reg(7);
            dato_tx_reg(5) <= dato_tx_reg(6);
            dato_tx_reg(4) <= dato_tx_reg(5);
            dato_tx_reg(3) <= dato_tx_reg(4);
            dato_tx_reg(2) <= dato_tx_reg(3);
            dato_tx_reg(1) <= dato_tx_reg(2);
            dato_tx_reg(0) <= dato_tx_reg(1);
        end if;
        dato_tx <= dato_tx_reg(0);
    end if;
end process;

end Behavioral;
```

Módulo Selección

```
-- Este bloque es el encargado de transmitir los datos que correspondan a la linea de transmision
-- según el estado en el que se encuentre.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Seleccion is
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           estado_tx : in STD_LOGIC_VECTOR (1 downto 0);
           dato_tx : in STD_LOGIC;
           fpga_tx : out STD_LOGIC);
end Seleccion;

architecture Behavioral of Seleccion is

begin

process (clk, reset, estado_tx, dato_tx)
begin
    if reset = '1' then
        fpga_tx <= '1';
    elsif clk'event and clk = '1' then
        case estado_tx is
            when "00" => fpga_tx <= '1';        -- Estado reposo
            when "01" => fpga_tx <= '0';        -- Estado bit de inicio
            when "10" => fpga_tx <= dato_tx;    -- Estado bits de datos
            when others => fpga_tx <= '1';      -- Estado bit final
        end case;
    end if;

end process;

end Behavioral;
```

Módulo UART_RX

```
-- Este bloque se corresponde al Top level del receptor del modulo UART

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity UART_RX is
    Port ( clk : in STD_LOGIC;  -- Señal de reloj de la placa.
           reset : in STD_LOGIC;    -- Señal de reset asincrono
           fpga_rx : in STD_LOGIC;  -- Trama que se recibe en serie desde el ordenador
           dato_rx_out : out STD_LOGIC_VECTOR (7 downto 0);  -- Proporciona los 8 bits del dato que se ha recibido. Este dato solo sera valido desde que aviso_rx valga '1' y mientras recibiendo sea '0'
           aviso_rx : out STD_LOGIC;    -- Aviso de que se ha recibido un nuevo dato y que esta disponible en dato_rx_out. El aviso se dara poniendo la señal a '1' durante un unico ciclo de reloj
           recibiendo : out STD_LOGIC); -- Cuando vale '1' indica que el modulo se encuentra recibiendo una trama y por tanto el valor del dato dato_rx_out no es valido
end UART_RX;

architecture Behavioral of UART_RX is

component RegIn
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           fpga_rx : in STD_LOGIC;
           fpga_rx_rg : out STD_LOGIC);
    end component;

component DivFreq_rx
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           en_divfreq : in STD_LOGIC;
           baud : out STD_LOGIC;
           baud_medio : out STD_LOGIC);
    end component;
component Control_rx
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           fpga_rx_rg : in STD_LOGIC;
           baud : in STD_LOGIC;
           baud_medio : in STD_LOGIC;
           fin_cont8bits : in STD_LOGIC;
           en_divfreq : out STD_LOGIC;
           recibiendo : out STD_LOGIC;
           aviso_rx : out STD_LOGIC;
           estado_tx : out STD_LOGIC_VECTOR (1 downto 0);
           desplaza : out STD_LOGIC);
    end component;

component CuentaBits
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           baud : in STD_LOGIC;
           estado_tx : in STD_LOGIC_VECTOR (1 downto 0);
           fin_cont8bits : out STD_LOGIC);
    end component;

component Desplz_SIPO
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           desplaza : in STD_LOGIC;
           fpga_rx_rg : in STD_LOGIC;
           dato_rx_out : out STD_LOGIC_VECTOR (7 downto 0));
    end component;

signal en_divfreq, baud, baud_medio, fin_cont8bits, desplaza, fpga_rx_rg : std_logic;
signal estado_tx : std_logic_vector(1 downto 0);

begin
RegIn_unit : RegIn
port map (
   clk => clk,
   reset => reset,
   fpga_rx => fpga_rx,
   fpga_rx_rg => fpga_rx_rg
);

DivFreq_rx_unit : DivFreq_rx
port map (
   clk => clk,
   reset => reset,
   en_divfreq => en_divfreq,
   baud => baud,
   baud_medio => baud_medio
);

Control_rx_unit : Control_rx
port map (
   clk => clk,
   reset => reset,
   fpga_rx_rg => fpga_rx_rg,
   baud => baud,
   baud_medio => baud_medio,
   fin_cont8bits => fin_cont8bits,
   en_divfreq => en_divfreq,
   recibiendo => recibiendo,
   aviso_rx => aviso_rx,
   estado_tx => estado_tx,
   desplaza => desplaza
);
CuentaBits_unit : CuentaBits
port map (
   clk => clk,
   reset => reset,
   baud => baud,
   estado_tx => estado_tx,
   fin_cont8bits => fin_cont8bits
);

Desplz_SIPO_unit : Desplz_SIPO
port map (
   clk => clk,
   reset => reset,
   desplaza => desplaza,
   fpga_rx_rg => fpga_rx_rg,
   dato_rx_out => dato_rx_out
);

end Behavioral;
```

Módulo RegIn

```
-- Este bloque se encarga de registrar la señal fpga_rx. Como esta señal es una señal asincrona, es conveniente registrarla.
-- Se ha registrado dos veces para evitar meta-estabilidad.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity RegIn is
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           fpga_rx : in STD_LOGIC;
           fpga_rx_rg : out STD_LOGIC);
end RegIn;

architecture Behavioral of RegIn is

    signal aux : std_logic;

begin

-- Para registrar dos veces, se colocan dos biestables en cascada.
    P_DETECTA: process (reset, clk)
    begin
        if reset = '1' then
            aux <= '1';
            fpga_rx_rg <= '1';
        elsif clk'event and clk = '1' then
            aux <= fpga_rx;
            fpga_rx_rg <= aux;
        end if;
    end process;

end Behavioral;
```

Módulo DivFreq_rx

```
-- Este bloque genera la señal ('baud') que indica cuándo ha pasado el intervalo de tiempo correspondiente a cada bit de la trama de recepcion.
-- Esta señal tendra una frecuencia determinada por la constante c_baud y que se corresponde con los baudios a los que se comunica la UART.

-- El funcionamiento es el mismo que para el transmisor, pero con la diferencia de que aqui se saca una señal nueva (baud_medio) que indica
-- el punto medio de cada bit de la transmisión, donde se evalúa el valor del bit recibido evitando así evaluar el bit cerca de las transiciones
-- donde se puede tomar el valor del bit contiguo.

library IEEE, WORK;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_arith.ALL;
use WORK.UART_PKG.ALL;

entity DivFreq_rx is
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           en_divfreq : in STD_LOGIC;
           baud : out STD_LOGIC;
           baud_medio : out STD_LOGIC);
end DivFreq_rx;

architecture Behavioral of DivFreq_rx is

signal cont_baud : unsigned (c_nb_cont_baud downto 0);

begin
P_DivFreq: Process(reset,clk)
begin
    if reset = '1' then
        cont_baud <= (others => '0');
        baud <= '0';
        baud_medio <= '0';
    elsif clk'event and clk='1' then
        if en_divfreq = '1' then
            baud <= '0';
            baud_medio <= '0';
            if cont_baud = c_fin_cont_baud then             -- Final de la cuenta -> baud = '1'
                baud <= '1';
                cont_baud <= (others => '0');
            elsif cont_baud = c_fin_cont_baud_medio then    -- Mitad de la cuenta -> baud_medio = '1'
                baud_medio <= '1';
                cont_baud <= cont_baud + 1;
            else
                cont_baud <= cont_baud + 1;
            end if;
        else
            cont_baud <= (others => '0');
        end if;
    end if;
end process;

end Behavioral;
```

Módulo Control_rx

```
-- Este modulo se encarga de dirigir al resto de bloques. Es el que contiene la máquina de estados, de tipo Mealy,
-- que determina el funcionamiento del circuito de recepción del modulo UART.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Control_rx is
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           fpga_rx_rg : in STD_LOGIC;
           baud : in STD_LOGIC;
           baud_medio : in STD_LOGIC;
           fin_cont8bits : in STD_LOGIC;
           en_divfreq : out STD_LOGIC;
           recibiendo : out STD_LOGIC;
           aviso_rx : out STD_LOGIC;
           estado_tx : out STD_LOGIC_VECTOR (1 downto 0);
           desplaza : out STD_LOGIC);
end Control_rx;

architecture Behavioral of Control_rx is

signal aux : std_logic;

-- Se definen los cuatro estados:
-- · e_init. El receptor se encuentra esperando la orden de recibir.
-- · e_bit_init. Se recibe el bit de inicio (linea a cero).
-- · e_bits_dato. Se reciben los 8 bits de datos.
-- · e_bit_fin. Se recibe el bit de fin (linea de recepcion a uno).
type estados_TX is (e_init, e_bit_init, e_bits_dato, e_bit_fin);
signal estado_actual, estado_siguiente : estados_TX;

begin
-- MAQUINA DE ESTADOS TIPO MEALY (PORQUE LAS SALIDAS DEPENDEN DE LAS ENTRADAS)

-- En este proceso se obtiene el estado siguiente a partir del estado actual y las entradas.
-- Es un proceso combinacional, por tanto, no depende de la señal de reloj ni de reset.
P_COMB_ESTADO : process(estado_actual, baud, fpga_rx_rg, fin_cont8bits)
begin
    case estado_actual is
        when e_init =>
            if fpga_rx_rg = '0' then                -- En este caso, cuando se inicia la recepcion es cuando llega un 0 en la linea rx.
                estado_siguiente <= e_bit_init;
            else
                estado_siguiente <= e_init;
            end if;
        when e_bit_init =>
            if baud = '1' then
                estado_siguiente <= e_bits_dato;
            else
                estado_siguiente <= e_bit_init;
            end if;
        when e_bits_dato =>
            if fin_cont8bits = '1' then
                estado_siguiente <= e_bit_fin;
            else
                estado_siguiente <= e_bits_dato;
            end if;
        when e_bit_fin =>
            if baud = '1' then
                estado_siguiente <= e_init;
            else
                estado_siguiente <= e_bit_fin;
            end if;
    end case;
end process;
-- Este proceso se encarga de actualizar el estado en cada ciclo de reloj y lo guarda en uno o varios
-- elementos de memoria. En cada ciclo de reloj, el estado siguiente pasa a ser el estado actual.
-- Como depende del reloj y de la señal de reset, es un proceso secuencial.
P_SEQ_FSM : process(clk, reset)
begin
    if reset = '1' then
        estado_actual <= e_init;
    elsif clk'event and clk = '1' then
        estado_actual <= estado_siguiente;
    end if;
end process;
-- Este proceso combinacional proporciona las salidas en funcion del estado actual y de las entradas, propio
-- de una máquina de estados tipo Mealy.
P_COM_SALIDAS : process(estado_actual, fin_cont8bits, baud, baud_medio)
begin
    case estado_actual is
        when e_init =>
            estado_tx <= "00";
            en_divfreq <= '0';
            recibiendo <= '0';
            aviso_rx <= '0';
            desplaza <= '0';
        when e_bit_init =>
            estado_tx <= "01";
            en_divfreq <= '1';
            recibiendo <= '1';      -- Esta señal actua igual que transmitiendo pero en la recepcion.
            aviso_rx <= '0';
            desplaza <= '0';
            if baud_medio = '1' then
                desplaza <= baud_medio;   -- Esta señal se utiliza para desplazar los bits en el registro de desplazamiento SIPO.
            else
                desplaza <= '0';
            end if;
        when e_bits_dato =>
            estado_tx <= "10";
            en_divfreq <= '1';
            recibiendo <= '1';
            aviso_rx <= '0';
            if fin_cont8bits = '0' then
                desplaza <= baud_medio;
            else
                desplaza <= '0';
            end if;
        when e_bit_fin =>
            estado_tx <= "11";
            en_divfreq <= '1';
            desplaza <= '0';
            if baud = '1' then
                aviso_rx <= '1';        -- Esta señal se pone a '1' cuando se termina de recibir toda la trama.
                recibiendo <= '0';
            else
                aviso_rx <= '0';
                recibiendo <= '1';
            end if;
    end case;
end process;

end Behavioral;
```

Módulo Desplz_SIPO

```
-- Este bloque se corresponde a un registro de desplazamiento al que se le van cargando los datos que le
-- llegan de la linea de recepción y los devuelve en paralelo. Es lo opuesto al registro del transmisor.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Desplz_SIPO is
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           desplaza : in STD_LOGIC;
           fpga_rx_rg : in STD_LOGIC;
           dato_rx_out : out STD_LOGIC_VECTOR (7 downto 0));
end Desplz_SIPO;

architecture Behavioral of Desplz_SIPO is

signal dato_rx_reg : std_logic_vector (7 downto 0) := (others =>'0');

begin
-- En este caso, la señal 'desplaza' tambien da la señal de que se almacene fpga_rx_rg en el bit MSB (Es decir, indica el inicio del registro).
P_registro : Process (clk, reset)
begin
    if reset = '1' then
        dato_rx_reg <= (others => '0');
    elsif clk'event and clk = '1' then
        if desplaza = '1' then
            dato_rx_reg(7) <= fpga_rx_rg;
            dato_rx_reg(6) <= dato_rx_reg(7);
            dato_rx_reg(5) <= dato_rx_reg(6);
            dato_rx_reg(4) <= dato_rx_reg(5);
            dato_rx_reg(3) <= dato_rx_reg(4);
            dato_rx_reg(2) <= dato_rx_reg(3);
            dato_rx_reg(1) <= dato_rx_reg(2);
            dato_rx_reg(0) <= dato_rx_reg(1);
        end if;
    end if;
end process;

dato_rx_out <= dato_rx_reg;

end Behavioral;
```

1300 Henley Court
Pullman, WA 99163
509.334.6306
www.digilentinc.com

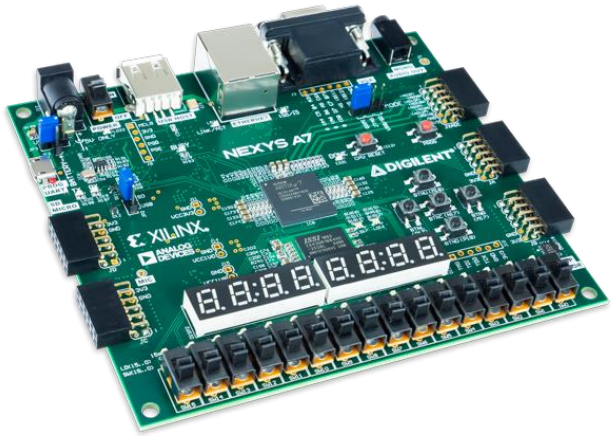
Nexys A7™ FPGA Board Reference Manual

Revised July 10, 2019

Table of Contents

Features

The Nexys A7 board is a complete, ready-to-use digital circuit development platform based on the latest Artix-7™ Field Programmable Gate Array (FPGA) from Xilinx®. With its large, high-capacity FPGA, generous external memories, and collection of USB, Ethernet, and other ports, the Nexys A7 can host designs ranging from introductory combinational circuits to powerful embedded processors. Several built-in peripherals, including an accelerometer, temperature sensor, MEMs digital microphone, a speaker amplifier, and several I/O devices allow the Nexys A7 to be used for a wide range of designs without needing any other components.



The Nexys A7 FPGA.

- **Artux-7 FPGA**
  - 15,850 Programmable logic slices, each with four 6-input LUTs and 8 flip-flops (*8,150 slices)
  - 1,188 Kbits of fast block RAM (*600 Kbits)
  - Six clock management tiles, each with phase-locked loop (PLL)
  - 240 DSP slices (*120 DSPs)
  - Internal clock speeds exceeding 450 MHz
  - Dual-channel, 1 MSPS internal analog-digital converter (XADC)
- **Memory**
  - 128MiB DDR2
  - Serial Flash
  - microSD card slot
- **Power**
  - Powered from USB or any 4.5V-5.5V external power source
- **USB and Ethernet**
  - 10/100 Ethernet PHY
  - USB-JTAG programming circuitry
  - USB-UART bridge
  - USB HID Host for mice, keyboards and memory sticks
- **Simple User Input/**Output
  - 16 Switches
  - 16 LEDs
  - Two RGB LEDs
  - Two 4-digit 7-segment displays
- **Audio and Video**
  - 12-bit VGA output
  - PWM audio output
  - PDM microphone
- **Additional Sensors**
  - 3-axis accelerometer
  - Temperature sensor
- **Expansion Connectors**
  - Pmod connector for XADC signals
  - Four Pmod connectors providing 32 total FPGA I/O

The Nexys A7-100T is compatible with Xilinx's Vivado® Design Suite as well as the ISE® toolset, which includes ChipScope™ and EDK. Xilinx ISE has been discontinued in favor of Vivado® Design Suite.

The Nexys A7-50T variant is compatible only with Vivado® Design Suite.

Xilinx offers free WebPACK™ versions of these toolsets, so designs can be implemented at no additional cost.

The Nexys A7 is not supported by the Digilent Adept Utility.

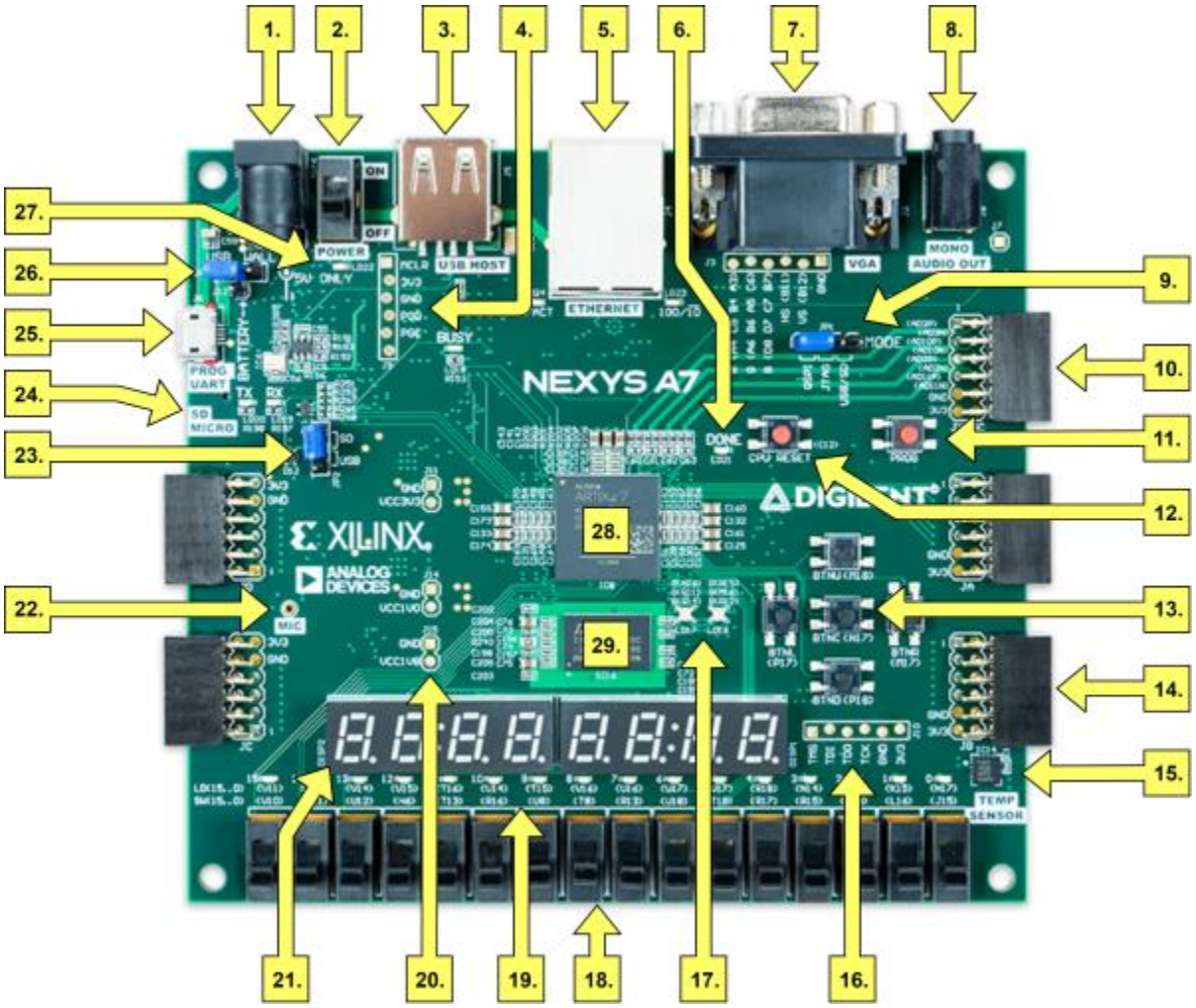


Figure 1. Nexys A7 Feature Callout.

Callout	Component Description	Callout	Component Description
1	Power jack	16	JTAG port for (optional) external cable
2	Power switch	17	Tri-color (RGB) LEDs
3	USB host connector	18	Slide switches (16)
4	PIC24 programming port (factory use)	19	LEDs (16)
5	Ethernet connector	20	Power supply test point(s)
6	FPGA programming done LED	21	Eight digit 7-seg display
7	VGA connector	22	Microphone
8	Audio connector	23	External configuration jumper (SD / USB)
9	Programming mode jumper	24	MicroSD card slot
10	Analog signal Pmod port (XADC)	25	Shared UART/ JTAG USB port

11	FPGA configuration reset button	**26**	Power select jumper and battery header
12	CPU reset button (for soft cores)	**27**	Power-good LED
13	Five pushbuttons	**28**	Xilinx Artix-7 FPGA
14	Pmod port(s)	**29**	DDR2 memory
15	Temperature sensor		

Purchasing Options

The Nexys A7 can be purchased with either a XC7A100T or XC7A50T FPGA loaded. These two Nexys A7 product variants are referred to as the Nexys A7-100T and Nexys A7-50T, respectively. When Digilent documentation describes functionality that is common to both of these variants, they are referred to collectively as the "Nexys A7". When describing something that is only common to a specific variant, the variant will be explicitly called out by its name.

The only difference between the Nexys A7-100T and Nexys A7-50T is the size of the Artix-7 part. The Artix-7 FPGAs both have the same capabilities, but the XC7100T has about a 2 times larger internal FPGA than the XC750T. The differences between the two variants are summarized below:

Product Variant	Nexys A7-100T	Nexys A7-50T
FPGA Part Number	XC7A100T-1CSG324C	XC7A50T-1CSG324I
Look-up Tables (LUTs)	63,400	32,600
Flip-Flops	126,800	65,200
Block RAM	1,188 Kb	600 Kb
DSP Slices	240	120
Clock Management Tiles	6	5

Board Revisions

The Nexys A7 is a rebrand of the Nexys 4 DDR board, which is an incremental update to the Nexys 4 board.

Migrating from Nexys 4 DDR

The only difference between the Nexys A7 and Nexys 4 DDR is the addition of the Nexys A7-50T variant of the Nexys A7, which has a smaller gate array. The Nexys A7-100T variant is functionally identical to the Nexys 4 DDR.

Users of the Nexys A7 may find resources produced for the Nexys 4 DDR helpful, which can be found at the Nexys 4 DDR's Resource Center.

Migrating from Nexys 4

The major improvement from the Nexys 4 to the Nexys 4 DDR is the replacement of the 16 MiB Cellular RAM with a 128 MiB DDR2 SDRAM memory. Furthermore, to accommodate the new memory, the pin-out of the FPGA banks changed as well.

The audio output (AUD_PWM) needs to be driven open-drain as opposed to push-pull on the Nexys 4.

1 Functional Description

1.1 Power Supplies

The Nexys A7 board can receive power from the Digilent USB-JTAG port (J6) or from an external power supply. Jumper JP3 (near the power jack) determines which source is used.

All Nexys A7 power supplies can be turned on and off by a single logic-level power switch (SW16). A power-good LED (LD22), driven by the "power good" output of the ADP2118 supply, indicates that the supplies are turned on and operating normally. An overview of the Nexys A7 power circuit is shown in Figure 1.1.

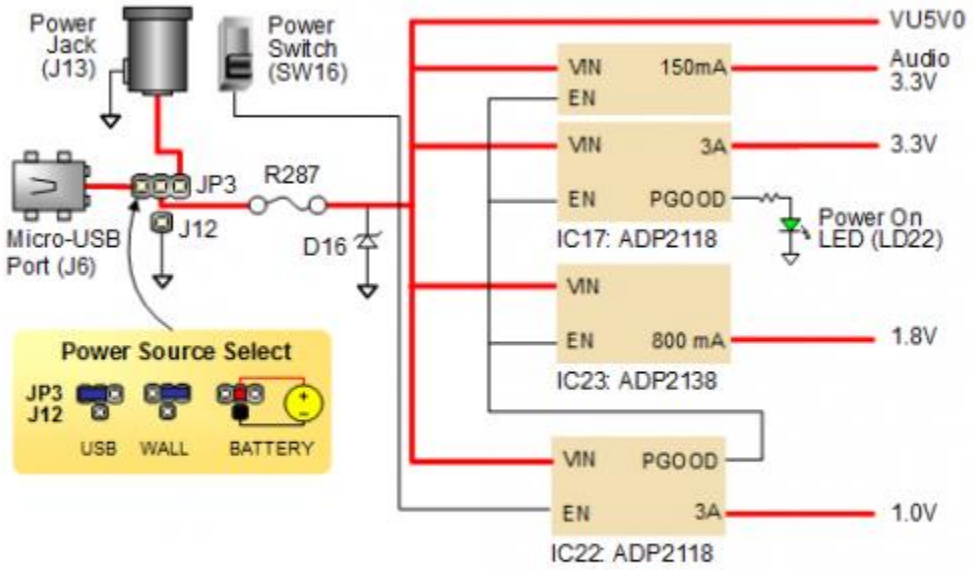


Figure 1.1 Nexys A7 Power Circuit

The USB port can deliver enough power for the vast majority of designs. In order to power the board from USB port set jumper JP3 to "USB". Our out-of-box demo draws ~400mA of current from the 5V input rail. A few demanding applications, including any that drive multiple peripheral boards, might require more power than the USB port can provide. Also, some applications may need to run without being connected to a PC's USB port. In these instances, an external power supply or battery pack can be used.

An external power supply can be used by plugging into to the power jack (J13) and setting jumper JP3 to "WALL". The supply must use a coax, center-positive 2.1mm internal-diameter plug, and deliver 4.5VDC to 5.5VDC and at least 1A of current (i.e., at least 5W of power). Many suitable supplies can be purchased from Digilent, through Digi-Key, or other catalog vendors.

An external battery pack can be used by connecting the battery's positive terminal to the center pin of JP3 and the negative terminal to the pin labeled J12, directly below JP3. Since the main regulator on the Nexys A7 cannot accommodate input voltages over 5.5VDC, an external battery pack must be limited to 5.5VDC. The minimum voltage of the battery pack depends on the application: if the USB Host function (J5) is used, at least 4.6V needs to be provided. In other cases, the minimum voltage is 3.6V.

Voltage regulator circuits from Analog Devices create the required 3.3V, 1.8V, and 1.0V supplies from the main power input. Table 1.1 provides additional information. Typical currents depend strongly on FPGA configuration and the values provided are typical of medium size/speed designs.

Supply	Circuits	Device	Current (Max/Typical)
3.3V	FPGA I/O, USB ports, Clocks, RAM I/O, Ethernet, SD slot, Sensors, Flash	IC17: ADP2118	3A/0.1 to 1.5A
1.0V	FPGA Core	IC22: ADP2118	3A/ 0.2 to 1.3A
1.8V	DDR2, FPGA Auxiliary and RAM	IC23: ADP2118	0.8A/ 0.5A

Table 1.1 Nexys A7 power supplies.

1.2 Protection

The Nexys A7 features overcurrent and overvoltage protection on the input power rail. A 3.5A fuse (R287) and a 5V Zener diode (D16) provide a non-resettable protection for other on-board integrated circuits, as displayed in Figure 2. Applying power outside of the specs outlined in this document is not covered by warranty. If this happens, either or both might get permanently damaged. The damaged parts are not user replaceable.

2 FPGA Configuration

After power-on, the Artix-7 FPGA must be configured (or programmed) before it can perform any functions. You can configure the FPGA in one of four ways:

1. A PC can use the Digilent USB-JTAG circuitry (portJ6, labeled "PROG") to program the FPGA any time the power is on.
2. A file stored in the nonvolatile serial (SPI) flash device can be transferred to the FPGA using the SPI port.
3. A programming file can be transferred to the FPGA from a micro SD card.
4. A programming file can be transferred from a USB memory stick attached to the USB HID port.

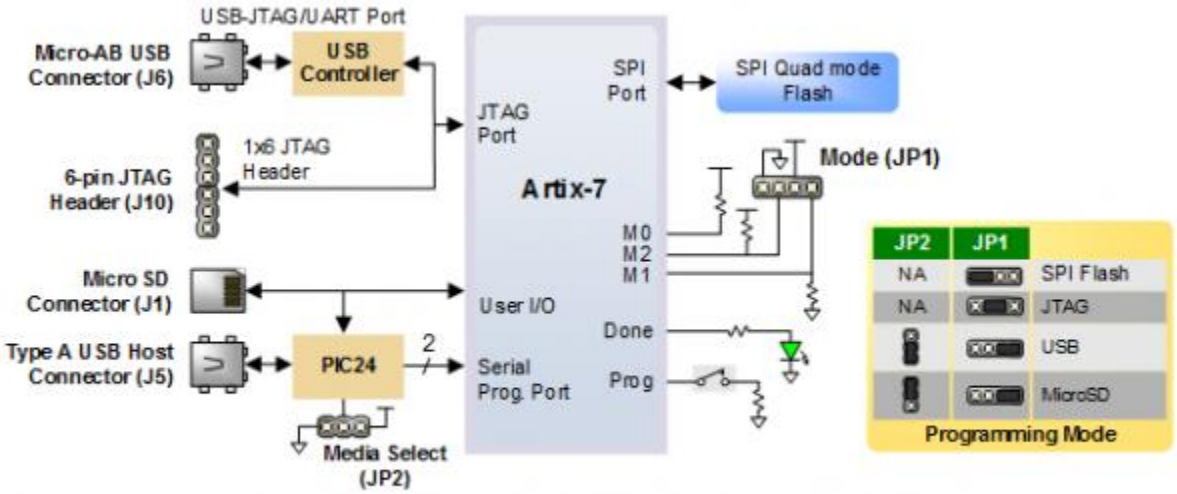


Figure 2.1 Nexys A7 DDR Configuration Options.

Figure 2.1 shows the different options available for configuring the FPGA. An on-board "mode" jumper (JP1) and a media selection jumper (JP2) select between the programming modes.

The FPGA configuration data is stored in files called bitstreams that have the .bit file extension. The ISE or Vivado software from Xilinx can create bitstreams from VHDL, Verilog®, or schematic-based source files (in the ISE toolset, EDK is used for MicroBlaze™ embedded processor-based designs).

Bitstreams are stored in SRAM-based memory cells within the FPGA. This data defines the FPGA's logic functions and circuit connections, and it remains valid until it is erased by removing board power, by pressing the reset button attached to the PROG input, or by writing a new configuration file using the JTAG port.

An Artix-7 100T bitstream is typically 30,606,304 bits and can take a long time to transfer. The time it takes to program the Nexys A7 can be decreased by compressing the bitstream before programming, and then allowing the FPGA to decompress the bitstream itself during configuration. Depending on design complexity, compression ratios of 10x can be achieved. Bitstream compression can be enabled within the Xilinx tools (ISE or Vivado) to occur during generation. For instructions on how to do this, consult the Xilinx documentation for the toolset being used. After being successfully programmed, the FPGA will cause the "DONE" LED to illuminate. Pressing the "PROG" button at any time will reset the configuration memory in the FPGA. After being reset, the FPGA will immediately attempt to reprogram itself from whatever method has been selected by the programming mode jumpers.

The following sections provide greater detail about programming the Nexys A7 using the different methods available.

2.1 JTAG Configuraiton

The Xilinx tools typically communicate with FPGAs using the Test Access Port and Boundary-Scan Architecture, commonly referred to as JTAG. During JTAG programming, a .bit file is transferred from the PC to the FPGA using the onboard Digilent USB-JTAG circuitry (port J6) or an external JTAG programmer, such as the Digilent JTAG-HS2, attached to port J10. You can perform JTAG programming any time after the Nexys A7 has been powered on, regardless of what the mode jumper (JP1) is set to. If the FPGA is already configured, then the existing configuration is overwritten with the bitstream being transmitted over JTAG. Setting the mode jumper to the JTAG setting (seen in Figure 3) is useful to prevent the FPGA from being configured from any other bitstream source until a JTAG programming occurs.

Programming the Nexys A7 with an uncompressed bitstream using the on-board USB-JTAG circuitry usually takes around five seconds. JTAG programming can be done using the hardware server in Vivado or the iMPACT tool included with ISE and the Lab Tools version of Vivado. The demonstration project available at http://www.digilentinc.com/ gives an in-depth tutorial on how to program your board.

2.2 Quad-SPI Configuration

Since the FPGA on the Nexys A7 is volatile, it relies on the Quad-SPI flash memory to store the configuration between power cycles. This configuration mode is called Master SPI. The blank FPGA takes the role of master and reads the configuration file out of the flash device upon power-up. To that effect, a configuration file needs to be downloaded first to the flash. When programming a nonvolatile flash device, a bitstream file is transferred to the flash in a two-step process. First, the FPGA is programmed with a circuit that can program flash devices, and then data is transferred to the flash device via the FPGA circuit (this complexity is hidden from the user by the Xilinx tools). This is called indirect programming. After the flash device has been programmed, it can automatically configure the FPGA at a subsequent power-on or reset event as determined by the mode jumper setting (see Figure 3). Programming files stored in the flash device will remain until they are overwritten, regardless of power-cycle events.

Programming the flash can take as long as four to five minutes, which is mostly due to the lengthy erase process inherent to the memory technology. Once written however, FPGA configuration can be very fast—less than a second. Bitstream compression, SPI bus width, and configuration rate are factors controlled by the Xilinx tools that can affect configuration speed. The Nexys A7 supports x1, x2, and x4 bus widths and data rates of up to 50 MHz for Quad-SPI programming.

Quad-SPI programming can be done using the iMPACT tool included with ISE or the Lab Tools version of Vivado.

2.3 USB Host and Micro SD Programming

You can program the FPGA from a pen drive attached to the USB Host port (J5) or a microSD card inserted into J1 by doing the following:

1. Format the storage device (Pen drive or microSD card) with a FAT32 file system.
2. Place a single .bit configuration file in the root directory of the storage device.
3. Attach the storage device to the Nexys A7.
4. Set the JP1 Programming Mode jumper on the Nexys A7 to “USB/SD”.
5. Select the desired storage device using JP2.
6. Push the PROG button or power-cycle the Nexys A7.

The FPGA will automatically configure with the .bit file on the selected storage device. Any .bit files that are not built for the proper Artix-7 device will be rejected by the FPGA.

The Auxiliary Function Status, or “BUSY” LED, gives visual feedback on the state of the configuration process when the FPGA is not yet programmed:

- When steadily lit, the auxiliary microcontroller is either booting up or currently reading the configuration medium (microSD or pen drive) and downloading a bitstream to the FPGA.
- A slow pulse means the microcontroller is waiting for a configuration medium to be plugged in.
- In case of an error during configuration, the LED will blink rapidly.

When the FPGA has been successfully configured, the behavior of the LED is application-specific. For example, if a USB keyboard is plugged in, a rapid blink will signal the receipt of an HID input report from the keyboard.

3 Memory

The Nexys A7 board contains two external memories: a 1Gib (128MiB) DDR2 SDRAM and a 128Mib (16MiB) non-volatile serial Flash device. The DDR2 modules are integrated on-board and connect to the FPGA using the industry standard interface. The serial Flash is on a dedicated quad-mode (x4) SPI bus. The connections and pin assignments between the FPGA and external memories are shown below.

3.1 DDR2

The Nexys A7 includes one Micron MT47H64M16HR-25:H DDR2 memory component, creating a single rank, 16-bit wide interface. It is routed to a 1.8V-powered HR (High Range) FPGA bank with 50 ohm controlled single-ended trace impedance. 50-ohm internal terminations in the FPGA are used to match the trace characteristics. Similarly, on the memory side, on-die terminations (ODT) are used for impedance matching.

For proper operation of the memory, a memory controller and physical layer (PHY) interface needs to be included in the FPGA design. There are two recommended ways to do that, which are outlined below and differ in complexity and design flexibility.

The straightforward way is to use the Digilent-provided DDR-to-SRAM adapter module which instantiates the memory controller and uses an asynchronous SRAM bus for interfacing with user logic. This module provides backward compatibility with projects written for older Nexys-line boards featuring a CellularRAM instead of DDR2. It trades memory bandwidth for simplicity.

More advanced users or those who wish to learn more about DDR SDRAM technology may want to use the Xilinx 7-series memory interface solutions core generated by the MIG (Memory Interface Generator) Wizard. Depending on the tool used (ISE, EDK or Vivado), the MIG Wizard can generate a native FIFO-style or an AXI4 interface to connect to user logic. This workflow allows the customization of several DDR parameters optimized for the particular application. Table 3.1 below lists the MIG Wizard settings optimized for the Nexys A7.

Setting	Value
Memory type	DDR2 SDRAM
Max. clock period	3000ps (667Mbps data rate)
Recommended clock period (for easy clock generation)	3077ps (650Mbps data rate)
Memory part	MT47H64M16HR-25E
Data width	16
Data mask	Enabled
Chip Select pin	Enabled
Rtt (nominal) – On-die termination	50ohms
Internal Vref	Enabled

Table 3.1.1 DDR2 settings for the Nexys A7.

Although the FPGA, memory IC, and the board itself are capable of the maximum data rate of 667Mbps, the limitations in the clock generation primitives restrict the clock frequencies that can be generated from the 100 MHz system clock. Thus, for simplicity, the next highest data rate of 650Mbps is recommended.

The MIG Wizard will require the fixed pin-out of the memory signals to be entered and validated before generating the IP core. For your convenience, an importable UCF file is provided on the Digilent website to speed up the process.

For more details on the Xilinx memory interface solutions, refer to the 7 Series FPGAs Memory Interface Solutions User Guide (ug586)[i].

3.2 Quad-SPI Flash

FPGA configuration files can be written to the Quad-SPI Flash (Spansion part number S25FL128S), and mode settings are available to cause the FPGA to automatically read a configuration from this device at power on. An Artix-7 100T configuration file requires just less than four MiB (mebibyte) of memory, leaving about 77% of the flash device available for user data. Or, if the FPGA is getting configured from another source, the whole memory can be used for custom data.

The contents of the memory can be manipulated by issuing certain commands on the SPI bus. The implementation of this protocol is outside the scope of this document. All signals in the SPI bus except SCK are general-purpose user I/O pins after FPGA configuration. SCK is an exception because it remains a dedicated pin even after configuration. Access to this pin is provided through a special FPGA primitive called STARTUPE2.

NOTE: Refer to the manufacturer's data sheets[ii] and Xilinx user guides[iii] for more information.

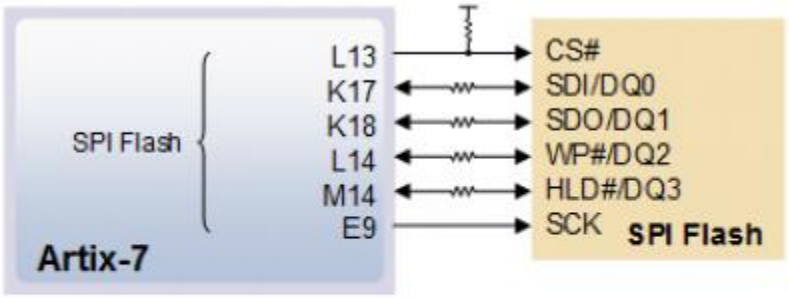


Figure 3.2.1 Nexys A7 DDR SPI Flash Pin-out

4 Ethernet PHY

The Nexys A7 board includes an SMSC 10/100 Ethernet PHY (SMSC part number LAN8720A) paired with an RJ-45 Ethernet jack with integrated magnetics. The SMSC PHY uses the RMII interface and supports 10/100 Mb/s. Figure 4.1 illustrates the pin connections between the Artix-7 and the Ethernet PHY. At power-on reset, the PHY is set to the following defaults:

- RMII mode interface
- Auto-negotiation enabled, advertising all 10/100 mode capable
- PHY address=00001

Two on-board LEDs (LD23 = LED2, LD24 = LED1) connected to the PHY provide link status and data activity feedback. See the PHY datasheet for details.

EDK-based designs can access the PHY using either the axi_ethernetlite (AXI EthernetLite) IP core or the axi_ethernet (Tri Mode Ethernet MAC) IP core. A mii_to_rmii core (Ethernet PHY MII to Reduced MII) needs to be inserted to convert the MAC interface from MII to RMII. Also, a 50 MHz clock needs to be generated for the

mii_to_rmii core and the CLKIN pin of the external PHY. To account for skew introduced by the mii_to_rmii core, generate each clock individually, with the external PHY clock having a 45 degree phase shift relative to the mii_to_rmii Ref_Clk. An EDK demonstration project that properly uses the Ethernet PHY can be found on the Nexys A7 product page at http://www.digilentinc.com/.

ISE designs can use the IP Core Generator wizard to create an Ethernet MAC controller IP core.

NOTE: Refer to the LAN8720A data sheet for further information[iv].

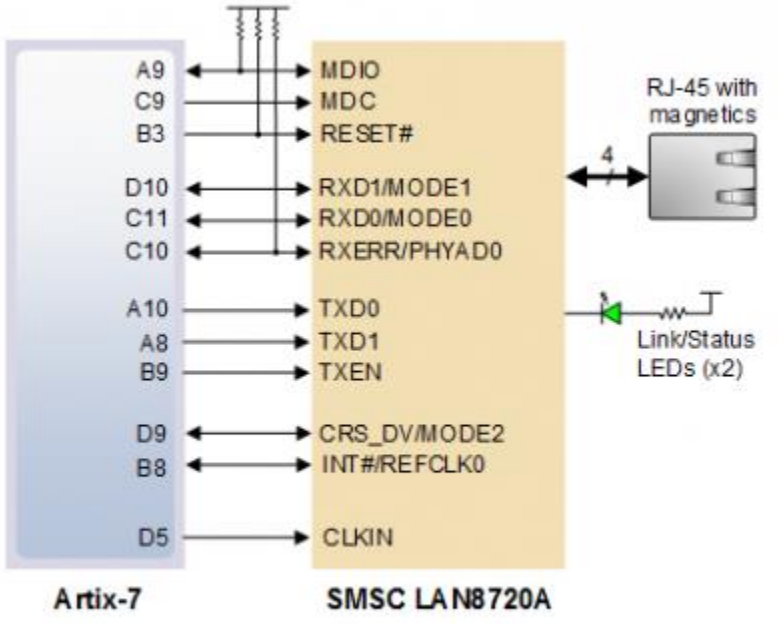


Figure 4.1 Pin Connections between the Artix-7 and the Ethernet PHY

5 Oscillators/Clocks

The Nexys A7 board includes a single 100 MHz crystal oscillator connected to pin E3 (E3 is a MRCC input on bank 35). The input clock can drive MMCMs or PLLs to generate clocks of various frequencies and with known phase relationships that may be needed throughout a design. Some rules restrict which MMCMs and PLLs may be driven by the 100 MHz input clock. For a full description of these rules and of the capabilities of the Artix-7 clocking resources, refer to the "7 Series FPGAs Clocking Resources User Guide" available from Xilinx.

Xilinx offers the Clocking Wizard IP core to help users generate the different clocks required for a specific design. This wizard will properly instantiate the needed MMCMs and PLLs based on the desired frequencies and phase relationships specified by the user. The wizard will then output an easy-to-use wrapper component around these clocking resources that can be inserted into the user's design. The clocking wizard can be accessed from within the Project Navigator or Core Generator tools.

6 USB-UART Bridge (Serial Port)

The Nexys A7 includes an FTDI FT2232HQ USB-UART bridge (attached to connector J6) that allows you use PC applications to communicate with the board using standard Windows COM port commands. Free USB-COM port drivers, available from http://www.ftdichip.com/ under the "Virtual Com Port" or VCP heading, convert USB packets to UART/serial port data. Serial port data is exchanged with the FPGA using a two-wire serial port (TXD/RXD) and optional hardware flow control (RTS/CTS). After the drivers are installed, I/O commands can be used from the PC directed to the COM port to produce serial data traffic on the C4 and D4 FPGA pins.

Two on-board status LEDs provide visual feedback on traffic flowing through the port: the transmit LED (LD20) and the receive LED (LD19). Signal names that imply direction are from the point-of-view of the DTE (Data Terminal Equipment), in this case the PC.

The FT2232HQ is also used as the controller for the Digilent USB-JTAG circuitry, but the USB-UART and USB-JTAG functions behave entirely independent of one another. Programmers interested in using the UART functionality of the FT2232 within their design do not need to worry about the JTAG circuitry interfering with the UART data transfers, and vice-versa. The combination of these two features into a single device allows the Nexys A7 to be programmed, communicated with via UART, and powered from a computer attached with a single Micro USB cable.

The connections between the FT2232HQ and the Artix-7 are shown in Figure 6.1.

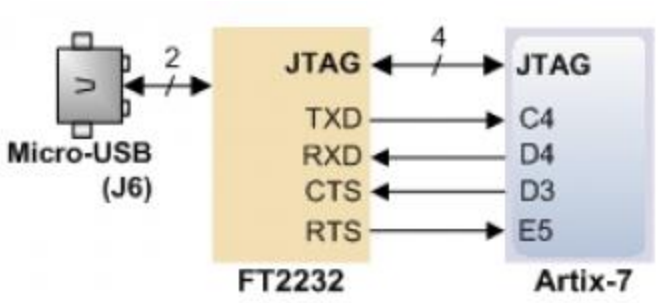


Figure 6.1 Nexys A7 FT2322HQ Connections

7 USB HID Host

The Auxiliary Function microcontroller (Microchip PIC24FJ128) provides the Nexys A7 with USB Embedded HID host capability. After power-up, the microcontroller is in configuration mode, either downloading a bitstream to the FPGA, or waiting to be programmed from other sources. Once the FPGA is programmed, the microcontroller switches to application mode, which is USB HID Host in this case. Firmware in the microcontroller can drive a mouse or a keyboard attached to the type A USB connector at J5 labeled "USB Host". Hub support is not currently available, so only a single mouse or a single keyboard can be used. Only keyboards and mice supporting the Boot HID interface are supported. The PIC24 drives several signals into the FPGA – two are used to implement a standard PS/2 interface for communication with a mouse or keyboard, and the others are connected to the FPGA's two-wire serial programming port, so the FPGA can be programmed from a file stored on a USB pen drive or microSD card.

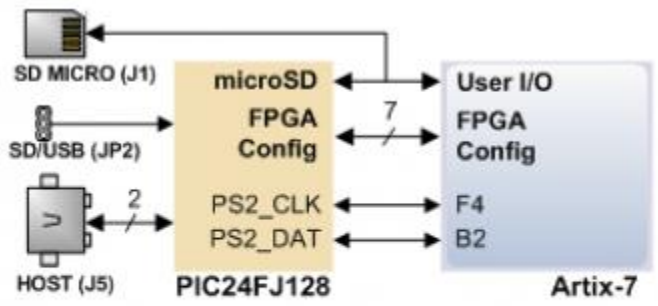
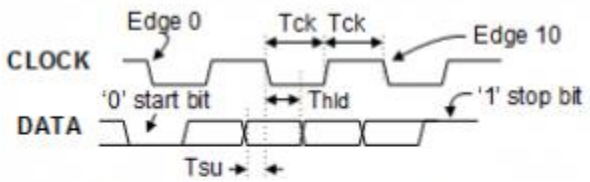


Figure 7.1 Nexys A7 PIC24 Connections

7.1 HID Controller

The Auxiliary Function microcontroller hides the USB HID protocol from the FPGA and emulates an old-style PS/2 bus. The microcontroller behaves just like a PS/2 keyboard or mouse would. This means new designs can re-use existing PS/2 IP cores. Mice and keyboards that use the PS/2 protocol use a two-wire serial bus (clock and data) to communicate with a host. On the Nexys A7, the microcontroller emulates a PS/2 device while the FPGA plays the role of the host. Both the mouse and the keyboard use 11-bit words that include a start bit, data byte (LSB first), odd parity, and stop bit, but the data packets are organized differently, and the keyboard interface allows bi-directional data transfers (so the host device can illuminate state LEDs on the keyboard). Bus timings are shown in Figure 7.1.1.



Symbol	Parameter	Min	Max
T_{CK}	Clock time	30us	50us
T_{SU}	Data-to-clock setup time	5us	25us
T_{HLD}	Clock-to-data hold time	5us	25us

Figure 7.1.1 PS/2 Device-to-Host Timing Diagram

The clock and data signals are only driven when data transfers occur; otherwise, they are held in the idle state at high-impedance (open-drain drivers). This requires that when the PS/2 signals are used in a design, internal pull-ups must be enabled in the FPGA on the data and clock pins. The clock signal is normally driven by the device, but may be held low by the host in special cases. The timings define signal requirements for mouse-to-host communications and bi-directional keyboard communications. A PS/2 interface circuit can be implemented in the FPGA to create a keyboard or mouse interface.

When a keyboard or mouse is connected to the Nexys A7, a “self-test passed” command (0xAA) is sent to the host. After this, commands may be issued to the device. Since both the keyboard and the mouse use the same PS/2 port, one can tell the type of device connected using the device ID. This ID can be read by issuing a Read ID

command (0xF2). Also, a mouse sends its ID (0x00) right after the “self-test passed” command, which distinguishes it from a keyboard.

7.2 Keyboard

PS/2-style keyboards use scan codes to communicate key press data. Each key is assigned a code that is sent whenever the key is pressed. If the key is held down, the scan code will be sent repeatedly about once every 100ms. When a key is released, an F0 key-up code is sent, followed by the scan code of the released key. If a key can be shifted to produce a new character (like a capital letter), then a shift character is sent in addition to the scan code and the host must determine which ASCII character to use. Some keys, called extended keys, send an E0 ahead of the scan code (and they may send more than one scan code). When an extended key is released, an E0 F0 key-up code is sent, followed by the scan code. Scan codes for most keys are shown in Figure 7.2.1.

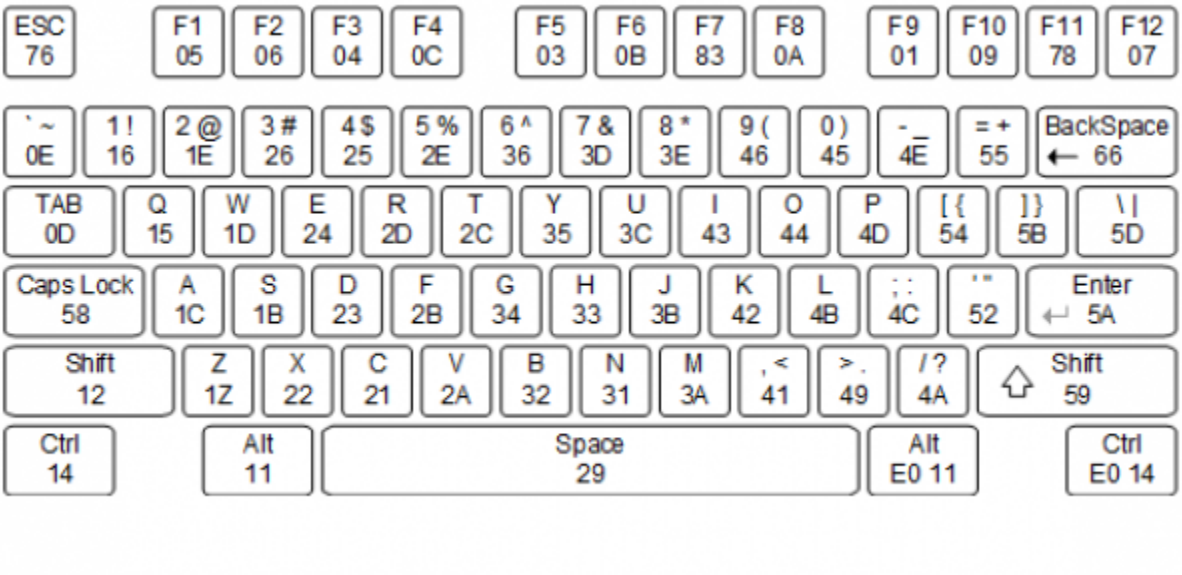


Figure 7.2.1 Keyboard Scan Codes

A host device can also send data to the keyboard. Table 7.2.1 shows a list of some common commands a host might send.

The keyboard can send data to the host only when both the data and clock lines are high (or idle). Because the host is the bus master, the keyboard must check to see whether the host is sending data before driving the bus. To facilitate this, the clock line is used as a “clear to send” signal. If the host drives the clock line low, the keyboard must not send any data until the clock is released. The keyboard sends data to the host in 11-bit words that contain a ‘0’ start bit, followed by 8-bits of scan code (LSB first), followed by an odd parity bit, and terminated with a ‘1’ stop bit. The keyboard generates 11 clock transitions (at 20 to 30 KHz) when the data is sent, and data is valid on the falling edge of the clock.

Command	Action
ED	Set Num Lock, Caps Lock, and Scroll Lock LEDs. Keyboard returns FA after receiving ED, then host sends a byte to set LED status: bit 0 sets Scroll Lock, bit 1 sets Num Lock, and bit 2 sets Caps lock. Bits 3 to 7 are ignored.
EE	Echo (test). Keyboard returns EE after receiving EE
F3	Set scan code repeat rate. Keyboard returns F3 on receiving FA, then host sends second byte to set the repeat rate.
FE	Resend. FE directs keyboard to re-send most recent scan code.

Table 7.2.1. Keyboard commands.

7.3 Mouse

Once entered in stream mode and data reporting is enabled, the mouse outputs a clock and data signal when it is moved; otherwise, these signals remain at logic '1.' Each time the mouse is moved, three 11-bit words are sent from the mouse to the host device, as shown in Figure 7.3.1. Each of the 11-bit words contains a '0' start bit, followed by 8 bits of data (LSB first), followed by an odd parity bit, and terminated with a '1' stop bit. Thus, each data transmission contains 33 bits, where bits 0, 11, and 22 are '0' start bits, and bits 11, 21, and 33 are '1' stop bits. The three 8-bit data fields contain movement data, as shown in Figure 7.3.1. Data is valid at the falling edge of the clock, and the clock period is 20 to 30 KHz.

The mouse assumes a relative coordinate system wherein moving the mouse to the right generates a positive number in the X field and moving to the left generates a negative number. Likewise, moving the mouse up generates a positive number in the Y field, and moving down represents a negative number (the XS and YS bits in the status byte are the sign bits – a '1' indicates a negative number). The magnitude of the X and Y numbers represent the rate of mouse movement; the larger the number, the faster the mouse is moving (the XV and YV bits in the status byte are movement overflow indicators. A '1' means overflow has occurred). If the mouse moves continuously, the 33-bit transmissions are repeated every 50ms or so. The L and R fields in the status byte indicate Left and Right button presses (a '1' indicates the button is being pressed).

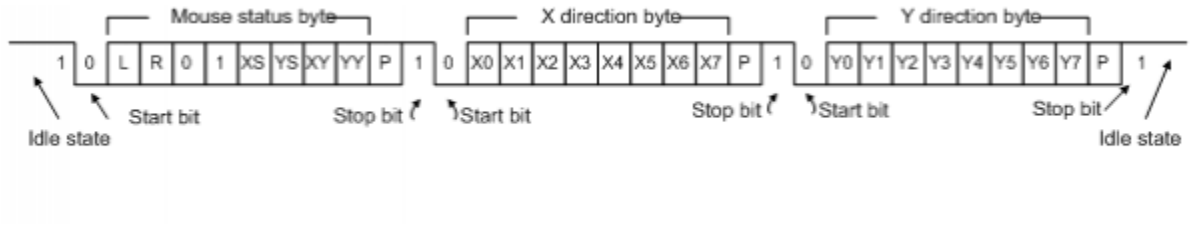


Figure 7.3.1 Mouse Data Format

The microcontroller also supports Microsoft® IntelliMouse®-type extensions for reporting back a third axis representing the mouse wheel, as shown in Table 7.3.1.

Command	Action
EA	Set stream mode. The mouse responds with "acknowledge" (0xFA) then resets its movement counters and enters stream mode.
F4	Enable data reporting. The mouse responds with "acknowledge" (0xFA) then enables data reporting and resets its movement counters. This command only affects behavior in stream mode. Once issued, mouse movement will automatically generate a data packet.
F5	Disable data reporting. The mouse responds with "acknowledge" (0xFA) then disables data reporting and resets its movement counters.
F3	Set mouse sample rate. The mouse responds with "acknowledge" (0xFA) then reads one more byte from the host. This byte is then saved as the new sample rate, and a new "acknowledge" packet is issued.
FE	Resend. FE directs mouse to re-send last packet.

Table 7.3.2. Microsoft IntelliMouse-Type extensions, commands, and actions.

8 VGA Port

The Nexys A7 board uses 14 FPGA signals to create a VGA port with 4 bits-per-color and the two standard sync signals (HS – Horizontal Sync, and VS – Vertical Sync). The color signals use resistor-divider circuits that work in conjunction with the 75-ohm termination resistance of the VGA display to create 16 signal levels each on the red, green, and blue VGA signals. This circuit, shown in Figure 8.1, produces video color signals that proceed in equal increments between 0V (fully off) and 0.7V (fully on). Using this circuit, 4096 different colors can be displayed, one for each unique 12-bit pattern. A video controller circuit must be created in the FPGA to drive the sync and color signals with the correct timing in order to produce a working display system.

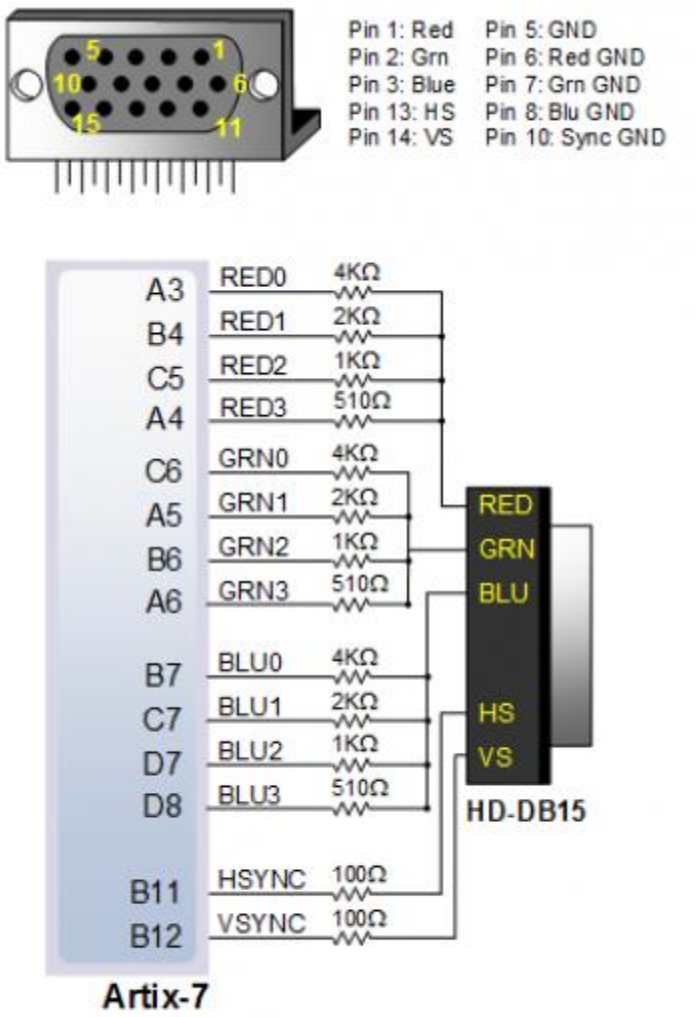


Figure 8.1 Nexys A7 VGA Interface

8.1 VGA System Timing

VGA signal timings are specified, published, copyrighted, and sold by the VESA® organization (http://www.vesa.org/). The following VGA system timing information is provided as an example of how a VGA monitor might be driven in 640 by 480 mode.

NOTE: For more precise information, or for information on other VGA frequencies, refer to documentation available at the VESA website.

CRT-based VGA displays use amplitude-modulated moving electron beams (or cathode rays) to display information on a phosphor-coated screen. LCD displays use an array of switches that can impose a voltage across a small amount of liquid crystal, thereby changing light permittivity through the crystal on a pixel-by-pixel basis. Although the following description is limited to CRT displays, LCD displays have evolved to use the same signal timings as CRT displays (so the "signals" discussion below pertains to both CRTs and LCDs). Color CRT displays use three electron beams (one for red, one for blue, and one for green) to energize the phosphor that coats the inner side of the display end of a cathode ray tube (see Figure 8.1.1).

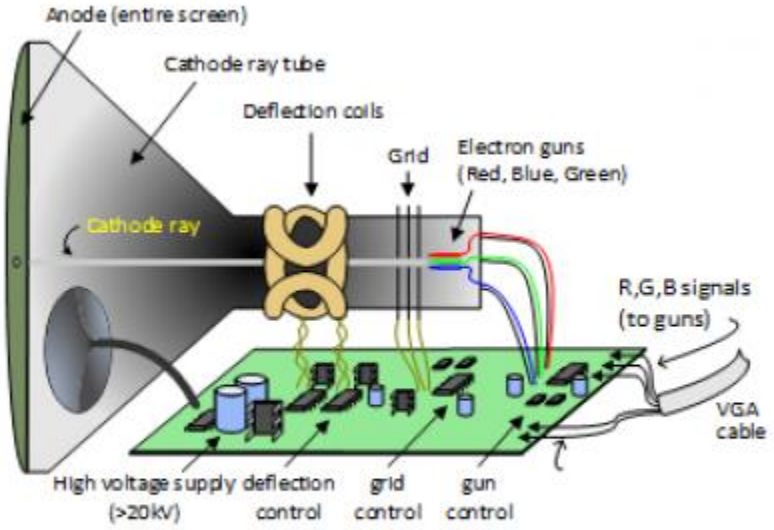


Figure 8.1.1 Color CRT Display

Electron beams emanate from "electron guns," which are finely-pointed heated cathodes placed in close proximity to a positively charged annular plate called a "grid." The electrostatic force imposed by the grid pulls rays of energized electrons from the cathodes, and those rays are fed by the current that flows into the cathodes. These particle rays are initially accelerated towards the grid, but they soon fall under the influence of the much larger electrostatic force that results from the entire phosphor-coated display surface of the CRT being charged to 20kV (or more). The rays are focused to a fine beam as they pass through the center of the grids, and then they accelerate to impact on the phosphor-coated display surface. The phosphor surface glows brightly at the impact point, and it continues to glow for several hundred microseconds after the beam is removed. The larger the current fed into the cathode, the brighter the phosphor will glow.

Between the grid and the display surface, the beam passes through the neck of the CRT where two coils of wire produce orthogonal electromagnetic fields. Because cathode rays are composed of charged particles (electrons), they can be deflected by these magnetic fields. Current waveforms are passed through the coils to produce magnetic fields that interact with the cathode rays and cause them to transverse the display surface in a "raster" pattern, horizontally from left to right and vertically from top to bottom, as shown in Figure 8.1.2. As the cathode ray moves over the surface of the display, the current sent to the electron guns can be increased or decreased to change the brightness of the display at the cathode ray impact point.

Information is only displayed when the beam is moving in the "forward" direction (left to right and top to bottom), and not during the time the beam is reset back to the left or top edge of the display. Much of the potential display time is therefore lost in "blanking" periods when the beam is reset and stabilized to begin a new horizontal or vertical display pass. The size of the beams, the frequency at which the beam can be traced across the display, and the frequency at which the electron beam can be modulated determine the display resolution.

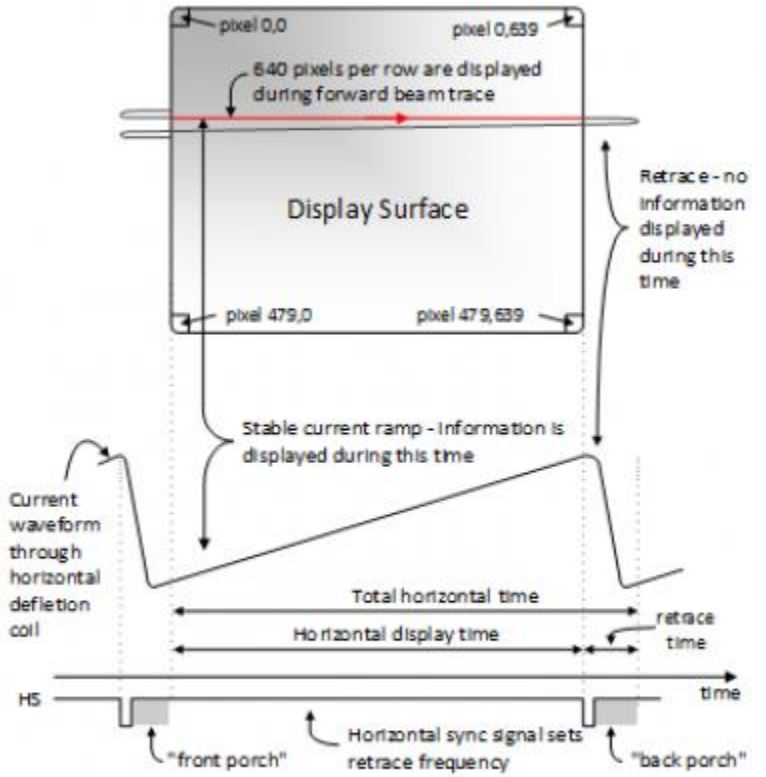


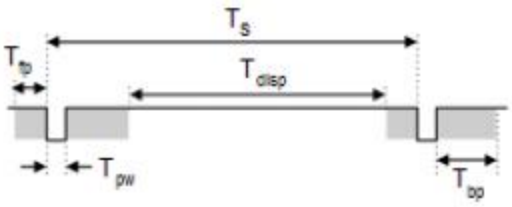
Figure 8.1.2 VGA Horizontal Synchronization

Modern VGA displays can accommodate different resolutions, and a VGA controller circuit dictates the resolution by producing timing signals to control the raster patterns. The controller must produce synchronizing pulses at 3.3V (or 5V) to set the frequency at which current flows through the deflection coils, and it must ensure that video data is applied to the electron guns at the correct time. Raster video displays define a number of "rows" that corresponds to the number of horizontal passes the cathode makes over the display area, and a number of "columns" that corresponds to an area on each row that is assigned to one "picture element," or pixel. Typical displays use from 240 to 1200 rows and from 320 to 1600 columns. The overall size of a display and the number of rows and columns determines the size of each pixel.

Video data typically comes from a video refresh memory; with one or more bytes assigned to each pixel location (the Nexys A7 uses 12 bits per pixel). The controller must index into video memory as the beams move across the display and retrieve and apply video data to the display at precisely the time the electron beam is moving across a given pixel.

A VGA controller circuit must generate the HS and VS timings signals and coordinate the delivery of video data based on the pixel clock. The pixel clock defines the time available to display one pixel of information. The VS signal defines the "refresh" frequency of the display, or the frequency at which all information on the display is redrawn. The minimum refresh frequency is a function of the display's phosphor and electron beam intensity, with practical

refresh frequencies falling in the 50Hz to 120Hz range. The number of lines to be displayed at a given refresh frequency defines the horizontal “retrace” frequency. For a 640-pixel by 480-row display using a 25 MHz pixel clock and 60 +/-1Hz refresh, the signal timings shown in Figure 8.1.3 can be derived. Timings for sync pulse width and front and back porch intervals (porch intervals are the pre- and post-sync pulse times during which information cannot be displayed) are based on observations taken from actual VGA displays.



Symbol	Parameter	Vertical Sync			Horiz. Sync	
		Time	Clocks	Lines	Time	Clks
T_S	Sync pulse	16.7ms	416,800	521	32 us	800
T_{disp}	Display time	15.36ms	384,000	480	25.6 us	640
T_{pw}	Pulse width	64 us	1,600	2	3.84 us	96
T_{fp}	Front porch	320 us	8,000	10	640 ns	16
T_{bp}	Back porch	928 us	23,200	29	1.92 us	48

Figure 8.1.3 Signal Timings for a 640-Pixel by 480-Row Display Using a 25 MHz Pixel Clock and 60 Hz Vertical Refresh

A VGA controller circuit, such as the one diagrammed in Figure 8.1.4, decodes the output of a horizontal-sync counter driven by the pixel clock to generate HS signal timings. You can use this counter to locate any pixel location on a given row. Likewise, the output of a vertical-sync counter that increments with each HS pulse can be used to generate VS signal timings, and you can use this counter to locate any given row. These two continually running counters can be used to form an address into video RAM. No time relationship between the onset of the HS pulse and the onset of the VS pulse is specified, so you can arrange the counters to easily form video RAM addresses, or to minimize decoding logic for sync pulse generation.

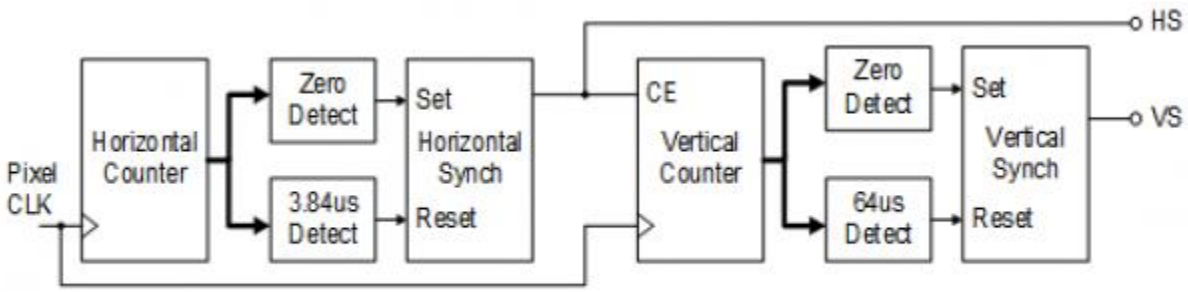


Figure 8.1.4 VGA Display Controller Block Diagram

9 Basic I/O

The Nexys A7 board includes two tri-color LEDs, sixteen slide switches, six push buttons, sixteen individual LEDs, and an eight-digit seven-segment display, as shown in Figure 9.1. The pushbuttons and slide switches are connected to the FPGA via series resistors to prevent damage from inadvertent short circuits (a short circuit could occur if an FPGA pin assigned to a pushbutton or slide switch was inadvertently defined as an output). The five pushbuttons arranged in a plus-sign configuration are "momentary" switches that normally generate a low output when they are at rest, and a high output only when they are pressed. The red pushbutton labeled "CPU RESET," on the other hand, generates a high output when at rest and a low output when pressed. The CPU RESET button is intended to be used in EDK designs to reset the processor, but you can also use it as a general-purpose pushbutton. Slide switches generate constant high or low inputs depending on their position.

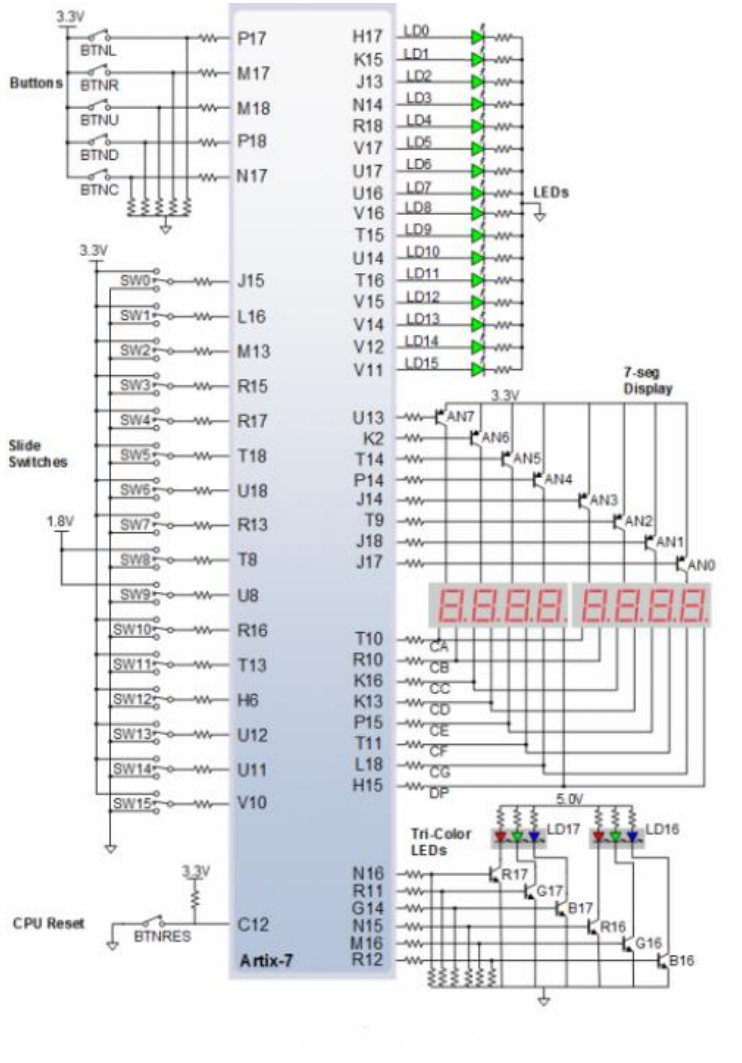


Figure 9.1 General Purpose I/O Devices on the Nexys A7

The sixteen-individual high-efficiency LEDs are anode-connected to the FPGA via 330-ohm resistors, so they will turn on when a logic high voltage is applied to their respective I/O pin. Additional LEDs that are not user-accessible indicate power-on, FPGA programming status, and USB and Ethernet port status.

9.1 Seven-Segment Display

The Nexys A7 board contains two four-digit common anode seven-segment LED displays, configured to behave like a single eight-digit display. Each of the eight digits is composed of seven segments arranged in a “figure 8” pattern, with an LED embedded in each segment. Segment LEDs can be individually illuminated, so any one of 128 patterns can be displayed on a digit by illuminating certain LED segments and leaving the others dark, as shown in Figure 9.1.1. Of these 128 possible patterns, the ten corresponding to the decimal digits are the most useful.

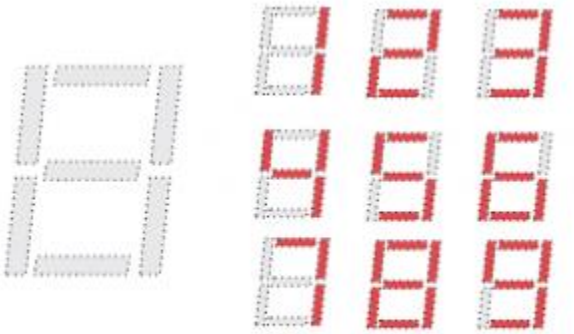


Figure 9.1.1 An Un-illuminated Seven-Segment Display and Nine Illumination Patterns Corresponding to Decimal Digits

The anodes of the seven LEDs forming each digit are tied together into one “common anode” circuit node, but the LED cathodes remain separate, as shown in Fig 18. The common anode signals are available as eight “digit enable” input signals to the 8-digit display. The cathodes of similar segments on all four displays are connected into seven circuit nodes labeled CA through CG. For example, the eight “D” cathodes from the eight digits are grouped together into a single circuit node called “CD.” These seven cathode signals are available as inputs to the 8-digit display. This signal connection scheme creates a multiplexed display, where the cathode signals are common to all digits but they can only illuminate the segments of the digit whose corresponding anode signal is asserted.

To illuminate a segment, the anode should be driven high while the cathode is driven low. However, since the Nexys A7 uses transistors to drive enough current into the common anode point, the anode enables are inverted. Therefore, both the AN0..7 and the CA..G/DP signals are driven low when active.

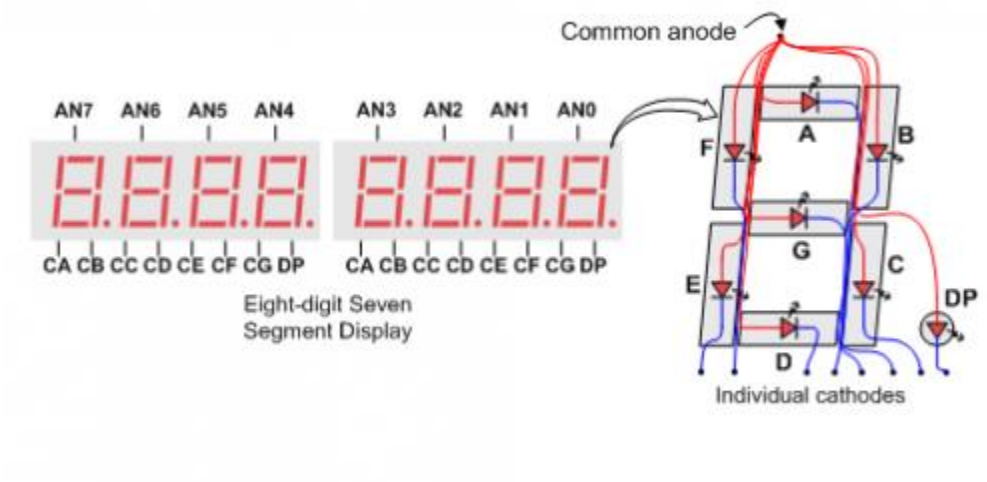


Figure 9.1.2 Common Anode Circuit Node

A scanning display controller circuit can be used to show an eight-digit number on this display. This circuit drives the anode signals and corresponding cathode patterns of each digit in a repeating, continuous succession at an update rate that is faster than the human eye can detect. Each digit is illuminated just one-eighth of the time, but because the eye cannot perceive the darkening of a digit before it is illuminated again, the digit appears continuously illuminated. If the update, or "refresh", rate is slowed to around 45Hz, a flicker can be noticed in the display.

For each of the four digits to appear bright and continuously illuminated, all eight digits should be driven once every 1 to 16ms, for a refresh frequency of about 1 KHz to 60Hz. For example, in a 62.5Hz refresh scheme, the entire display would be refreshed once every 16ms, and each digit would be illuminated for 1/8 of the refresh cycle, or 2ms. The controller must drive low the cathodes with the correct pattern when the corresponding anode signal is driven high. To illustrate the process, if AN0 is asserted while CB and CC are asserted, then a "1" will be displayed in digit position 1. Then, if AN1 is asserted while CA, CB, and CC are asserted, a "7" will be displayed in digit position 2. If AN0, CB, and CC are driven for 4ms, and then AN1, CA, CB, and CC are driven for 4ms in an endless succession, the display will show "71" in the first two digits. An example timing diagram for a four-digit controller is shown in Figure 9.1.3.

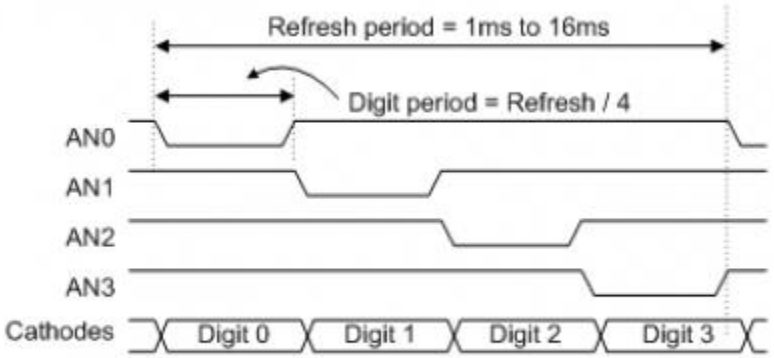


Figure 9.1.3 Four Digit Scanning Display Controller Timing Diagram

9.2 Tri-Color LED

The Nexys A7 board contains two tri-color LEDs. Each tri-color LED has three input signals that drive the cathodes of three smaller internal LEDs: one red, one blue, and one green. Driving the signal corresponding to one of these colors high will illuminate the internal LED. The input signals are driven by the FPGA through a transistor, which inverts the signals. Therefore, to light up the tri-color LED, the corresponding signals need to be driven high. The tri-color LED will emit a color dependent on the combination of internal LEDs that are currently being illuminated. For example, if the red and blue signals are driven high, and green is driven low, the tri-color LED will emit a purple color.

Note: Digilent strongly recommends the use of pulse-width modulation (PWM) when driving the tri-color LEDs (for information on PWM, see section 15.1 Pulse Density Modulation (PDM)). Driving any of the inputs to a steady logic '1' will result in the LED being illuminated at an uncomfortably bright level. You can avoid this by ensuring that none of the tri-color signals are driven with more than a 50% duty cycle. Using PWM also greatly expands the potential color palette of the tri-color led. Individually adjusting the duty cycle of each color between 50% and 0% causes the different colors to be illuminated at different intensities, allowing virtually any color to be displayed.

10 Pmod Ports

The Pmod ports are arranged in a 2×6 right-angle, and are 100-mil female connectors that mate with standard 2×6 pin headers. Each 12-pin Pmod port provides two 3.3V VCCsignals (pins 6 and 12), two Ground signals (pins 5 and 11), and eight logic signals, as shown in Figure 10.1. The VCC and Ground pins can deliver up to 1A of current. Pmod data signals are not matched pairs, and they are routed using best-available tracks without impedance control or delay matching. Pin assignments for the Pmod I/O connected to the FPGA are shown in Table 5.

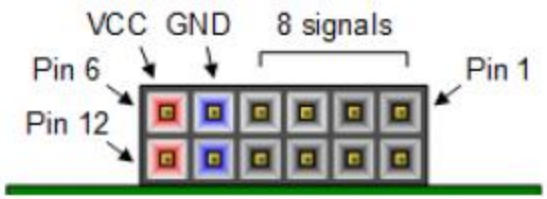


Figure 10.1 Pmod Connectors; Front View, as Loaded on PCB

Pmod JA	Pmod JB	Pmod JC	Pmod JD	Pmod XDAC
JA1: C17	JB1: D14	JC1: K1	JD1: H4	JXADC1: A13 (AD3P)
JA2: D18	JB2: F16	JC2: F6	JD2: H1	JXADC2: A15 (AD10P)
JA3: E18	JB3: G16	JC3: J2	JD3: G1	JXADC3: B16 (AD2P)
JA4: G17	JB4: H14	JC4: G6	JD4: G3	JXADC4: B18 (AD11P)
JA7: D17	JB7: E16	JC7: E7	JD7: H2	JXADC7: A14 (AD3N)
JA8: E17	JB8: F13	JC8: J3	JD8: G4	JXADC8: A16 (AD10N)
JA9: F18	JB9: G13	JC9: J4	JD9: G2	JXADC9: B17 (AD2N)

Table 10.1. Nexys A7 Pmod Pin Assignment

Digilent produces a large collection of Pmod accessory boards that can attach to the Pmod expansion connectors to add ready-made functions like A/D's, D/A's, motor drivers, sensors, as well as other functions.
See http://www.digilentinc.com/ for more information.

10.1 Dual Analog/Digital Pmod

The on-board Pmod expansion connector labeled "JXADC" is wired to the auxiliary analog input pins of the FPGA. Depending on the configuration, this connector can be used to input differential analog signals to the analog-to-digital converter inside of the Artix-7 (XADC). Any or all pairs in the connector can be configured either as analog input or digital input-output.

The Dual Analog/Digital Pmod on the Nexys A7 differs from the rest in the routing of its traces. The eight data signals are grouped into four pairs, with the pairs routed closely coupled for better analog noise immunity. Furthermore, each pair has a partially loaded anti-alias filter laid out on the PCB. The filter does not have capacitors C60-C63. In designs where such filters are desired, the capacitors can be manually loaded by the user.

NOTE: The coupled routing and the anti-alias filters might limit the data speeds when used for digital signals.

The XADC core within the Artix-7 is a dual channel 12-bit analog-to-digital converter capable of operating at 1 MSPS. Either channel can be driven by any of the auxiliary analog input pairs connected to the JXADC header. The XADC core is controlled and accessed from a user design via the Dynamic Reconfiguration Port (DRP). The DRP also provides access to voltage monitors that are present on each of the FPGA's power rails, and a temperature sensor

that is internal to the FPGA. For more information on using the XADC core, refer to the Xilinx document titled "7 Series FPGAs and Zynq-7000 All Programmable SoC XADC Dual 12-Bit 1 MSPS Analog-to-Digital Converter."

11 MicroSD Slot

The Nexys A7 provides a microSD slot for both FPGA configuration and user access. The on-board Auxiliary Function microcontroller shares the SD card bus with the FPGA. Before the FPGA is configured the microcontroller must have access to the SD card via SPI. Once a bit file is downloaded to the FPGA (from any source), the microcontroller power cycles the SD slot and relinquishes control of the bus. This enables any SD card in the slot to reset its internal state machines and boot up in SD native bus mode. All of the SD pins on the FPGA are wired to support full SD speeds in native interface mode, as shown in Figure 11.1. The SPI is also available, if needed. Once control over the SD bus is passed from the microcontroller to the FPGA, the SD_RESET signal needs to be actively driven low by the FPGA to power the microSD card slot. For information on implementing an SD card controller, refer to the SD card specification available at http://www.sdcard.org/

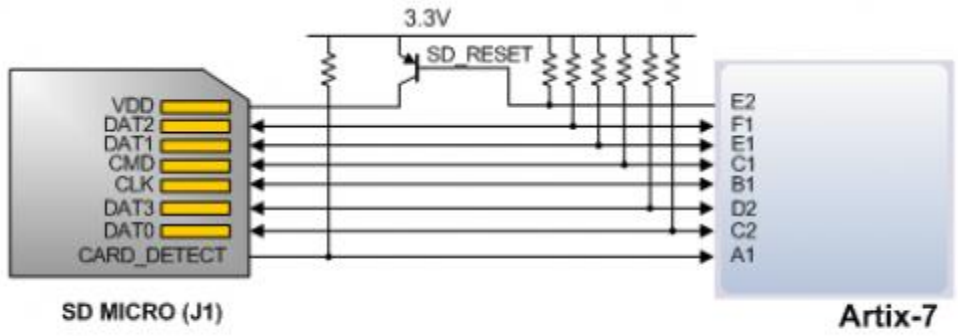


Figure 11.1 Artix-7 microSD Card Connector Interface (PIC24 Connections not Shown)

12 Temperature Sensor

The Nexys A7 includes an Analog Device ADT7420 temperature sensor. The sensor provides up to 16-bit resolution with a typical accuracy better than 0.25 degrees Celsius. The interface between the temperature sensor and FPGA is shown in Figure 12.1.

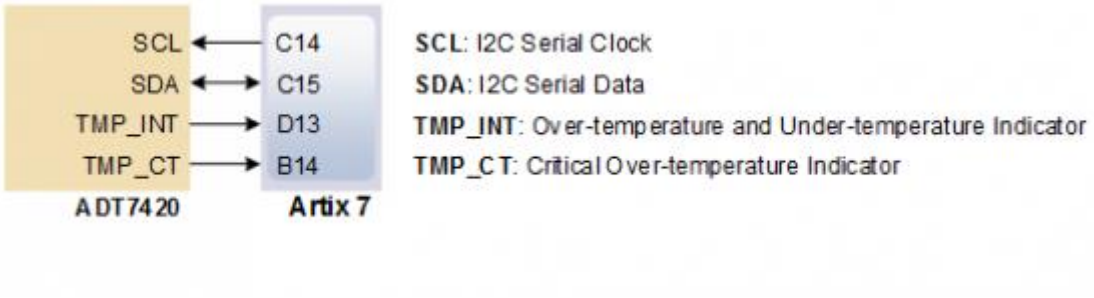


Figure 12.1 Temperature Sensor Interface

12.1 I²C Interface

The ADT7420 chip acts as a slave device using the industry standard I²C communication scheme. To communicate with ADT7420 chip, the I²C master must specify a slave address (0x4B) and a flag indicating whether the communication is a read (1) or a write (0). Once specifications are made for communication, a data transfer takes place. For ADT7420, the data transfer should consist of the address of the desired device register followed by the data to be written to the specified register. To read from a register, the master must write the desired register address to the ADT7420, then send an I²C restart condition, and send a new read request to the ADT7420. If the master does not generate a restart condition prior to attempting the read, the value written to the address register will be reset to 0x00. As some registers store 16-bit values as 8-bit register pairs, the ADT7420 will automatically increment the address register of the device when accessing certain registers, such as the temperature registers and the threshold registers. This allows for the master to use a single read or write request to access both the low and high bytes of these registers. A complete listing of registers and their behavior can be found in the ADT7420 datasheet available on the Analog Devices website.

12.2 Open Drain Outputs

The ADT7420 provides two open drain output signals to indicate when pre-set temperature thresholds are reached. If the temperature leaves a range defined by registers TLOW (0x06:0x07) and THIGH (0x04:0x05), the INT pin can be driven low or high based upon the configuration of the device. Similarly, the CT pin can be driven low or high if the temperature exceeds a critical threshold defined in TCRIT (0x08:0x09). Both of these pins need internal FPGA pull-ups when used.

For details on the electrical specifications and configuration of the INT and CT pins, refer to the ADT7420 datasheet.

12.3 Quick Start Operation

When the ADT7420 is powered up, it is in a mode that can be used as a simple temperature sensor without any initial configuration. By default, the device address register points to the temperature MSB register, so a two byte read without specifying a register will read the value of the temperature register from the device. The first byte read back will be the most significant byte (MSB) of the temperature data, and the second will be the least significant byte (LSB) of the data. These two bytes form a two's complement 16-bit integer. If the result is shifted to the right three bits and multiplied by 0.0625, the resulting signed floating point value will be a temperature reading in degrees Celsius.

For information on reading and writing to the other registers of the device, as well as notes on the accuracy of the temperature measurements, refer to the ADT7420 datasheet.

13 Accelerometer

The Nexys A7 includes an Analog Device ADXL362 accelerometer. The ADXL362 is a 3-axis MEMS accelerometer that consumes less than 2µA at a 100Hz output data rate and 270nA when in motion triggered wake-up mode. Unlike accelerometers that use power duty cycling to achieve low power consumption, the ADXL362 does not alias input signals by under-sampling; it samples the full bandwidth of the sensor at all data rates. The ADXL362 always provides 12-bit output resolution; 8-bit formatted data is also provided for more efficient single-byte transfers when a lower resolution is sufficient. Measurement ranges of ±2 g, ±4 g, and ±8 g are available with a resolution of 1 mg/LSB on the ±2 g range. The FPGA can talk with the ADXL362 via SPI interface. While the ADXL362 is in

Measurement Mode, it continuously measures and stores acceleration data in the X-data, Y-data, and Z-data registers. The interface between the FPGA and accelerometer can be seen in Figure 13.1.

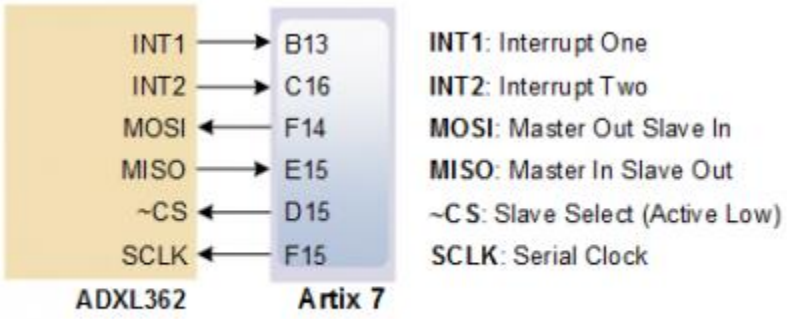


Figure 13.1 Accelerometer Interface

13.1 SPI Interface

The ADXL362 acts as a slave device using an SPI communication scheme. The recommended SPI clock frequency ranges from 1 MHz to 5 MHz. The SPI operates in SPI mode 0 with CPOL = 0 and CPHA = 0. All communications with the device must specify a register address and a flag that indicate whether the communication is a read or a write. Actual data transfer always follows the register address and communication flag. Device configuration can be performed by writing to the control registers within the accelerometer. Access accelerometer data by reading the device registers.

For a full list of registers, their functionality, and communication specifications, refer to the ADXL362 datasheet[v].

13.2 Interrupts

Several of the built-in functions of the ADXL362 can trigger interrupts that alert the host processor of certain status conditions. Interrupts can be mapped to either (or both) of two interrupt pins (INT1, INT2). Both of these pins require internal FPGA pull-ups when used. For more details about the interrupts, see the ADXL362 datasheet.

14 Microphone

The Nexys A7 board includes an omnidirectional MEMS microphone. The microphone uses an Analog Device ADMP421 chip which has a high signal to noise ratio (SNR) of 61dBA and high sensitivity of -26 dBFS. It also has a flat frequency response ranging from 100Hz to 15 kHz. The digitized audio is output in the pulse density modulated (PDM) format. The component architecture is shown in Figure 14.1.

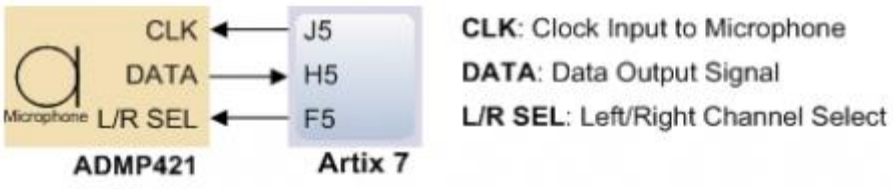


Figure 14.1 Microphone Block Diagram

14.1 Pulse Density Modulation (PDM)

PDM data connections are becoming more and more popular in portable audio applications, such as cellphones and tablets. With PDM, two channels can be transmitted with only two wires. The frequency of a PDM signal usually falls in the range of 1 MHz to 3 MHz. In a PDM bitstream, a 1 corresponds to a positive pulse and a 0 corresponds to a negative pulse. A run consisting of all '1's would correspond to the maximum positive value and a run of '0's would correspond to the minimum amplitude value. Figure 14.1.1 shows how a sine wave is represented in PDM signal.

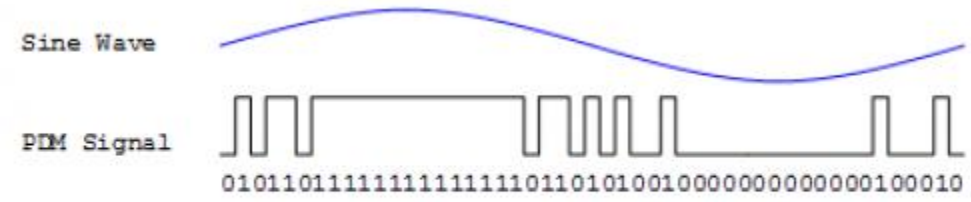


Figure 14.1.1 PDM Representation of a Sine Wave

A PDM signal is generated from an analog signal through a process called delta-sigma modulation. A simple idealized circuit of delta-sigma modulator is shown in Figure 14.1.2.

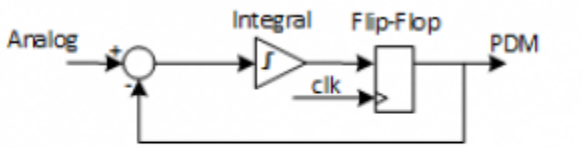


Figure 14.1.2 Simple Delta-Sigma Modulator Circuit

Sum	Integrator Out	Flip-flop Output
0.4-0=0.4	0+0.4=0.4	0
0.4-0=0.4	0.4+0.4=0.8	1
0.4-1=-0.6	0.8-0.6=0.2	0
0.4-0=0.4	0.2+0.4=0.6	1
0.4-1=-0.6	0.6-0.6=0	0
0.4-0=0.4	0+0.4=0.4	0
0.4-0=0.4	0.4+0.4=0.8	1

Table 14.1.2. Sigma delta modulator with a 0.4Vdd input.

To keep things simple, assume that the analog input and digital output have the same voltage range 0~Vdd. The input of the flip-flop acts like a comparator (any signal above Vdd/2 is considered as '1' and any input bellow Vdd/2 is considered '0'). The input of the integral circuit is the difference of the input analog signal and the PDM signal of the previous clock cycle. The integral circuit then integrates both of these inputs, and the output of the integral circuit is sampled by a D-Flip-flop. Table 6 shows the function of the delta-sigma modulator with an input of 0.4Vdd.

Note that the average of the flip-flop output equals the value of the input analog signal. So in order to get the value of analog input, all that is needed is a counter that counts the '1's for a certain period of time.

14.2 Microphone Digital Interface Timing

The clock input of the microphone can range from 1 MHz to 3.3 MHz based on the sampling rate and data precision requirement of the applications. The L/R Select signal must be set to a valid level, depending on which edge of the clock the data bit will be read. A low level on L/RSEL makes data available on the rising edge of the clock, while a high level corresponds to the falling edge of the clock, as shown in Figure 14.2.1.

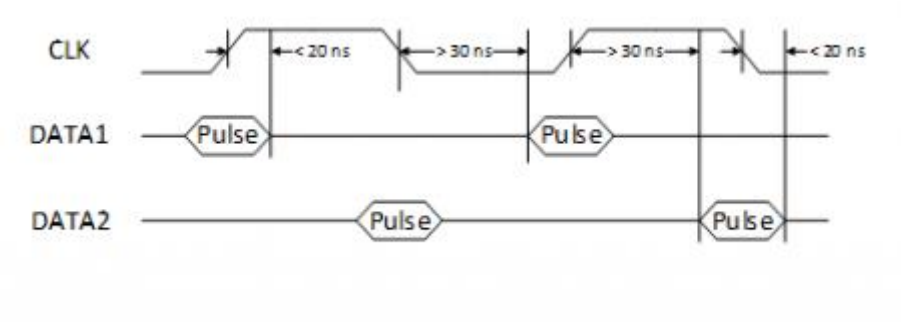


Figure 14.2.1 PDM Timing Diagram

The typical value of the clock frequency is 2.4 MHz. Assuming that the application requires 7-bit precision and 24 KHz, there can be two counters that count 128 samples at 12 KHz, as shown in Figure 14.2.2.

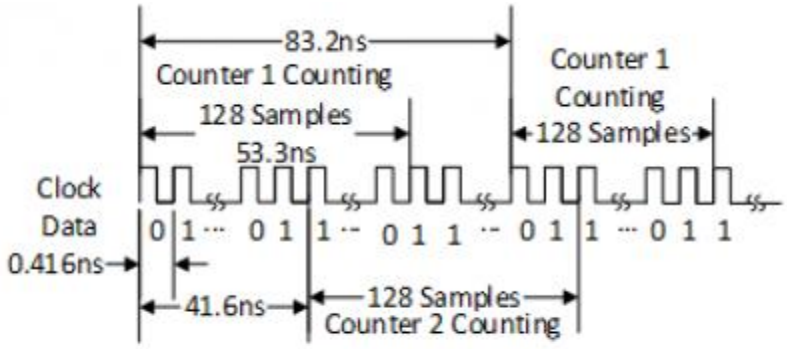


Figure 14.2.2 Sampling PDM with Two Counters

15 Mono Audio Output

The on-board audio jack (J8) is driven by a Sallen-Key Butterworth Low-pass 4th Order Filter that provides mono audio output. The circuit of the low-pass filter is shown in Figure 15.1. The input of the filter (AUD_PWM) is connected to the FPGA pin A11. A digital input will typically be a pulse-width modulated (PWM) or pulse density modulated (PDM) open-drain signal produced by the FPGA. The signal needs to be driven low for logic '0' and left in high-impedance for logic '1'. An on-board pull-up resistor to a clean analog 3.3V rail will establish the proper voltage for logic '1'. The low-pass filter on the input will act as a reconstruction filter to convert the pulse-width modulated digital signal into an analog voltage on the audio jack output.

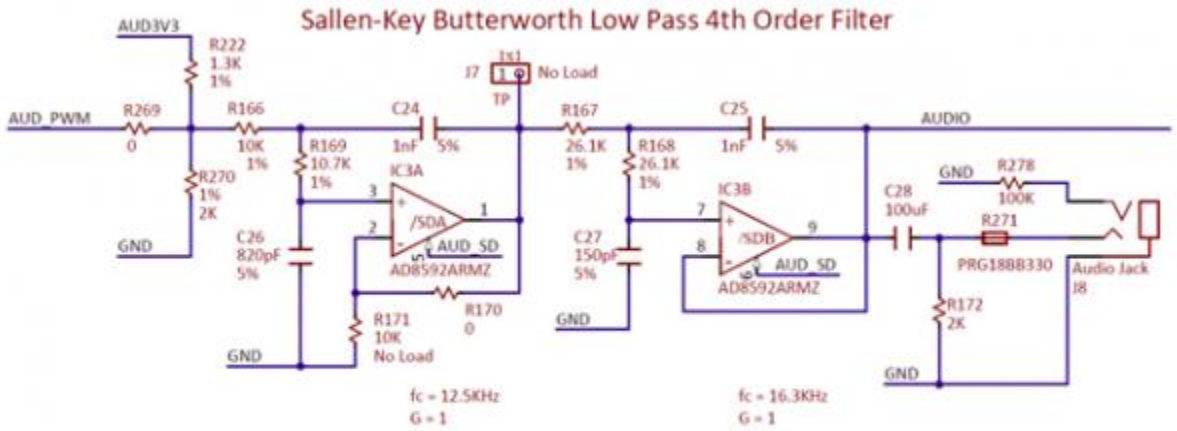


Figure 15.1 Sallen-Key Butterworth Low-Pass 4th Order Filter

The frequency response of SK Butterworth Low-Pass Filter is shown in Figure 15.2. The AC analysis of the circuit is done using NI Multisim 12.0.

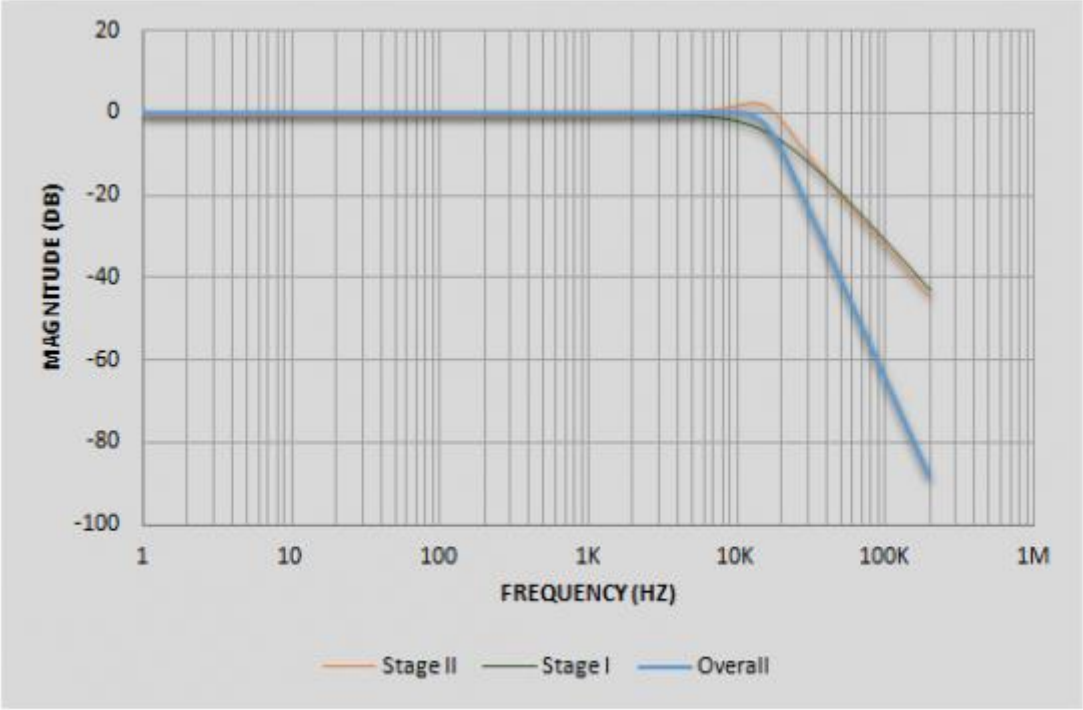


Figure 15.2 SK Butterworth Low-Pass Filter Frequency Response

15.2 Pulse-Width Modulation

A pulse-width modulated (PWM) signal is a chain of pulses at some fixed frequency, with each pulse potentially having a different width. This digital signal can be passed through a simple low-pass filter that integrates the digital waveform to produce an analog voltage proportional to the average pulse-width over some interval (the interval is

determined by the 3dB cut-off frequency of the low-pass filter and the pulse frequency). For example, if the pulses are high for an average of 10% of the available pulse period, then an integrator will produce an analog value that is 10% of the Vdd voltage. Figure 15.1.1 shows a waveform represented as a PWM signal.

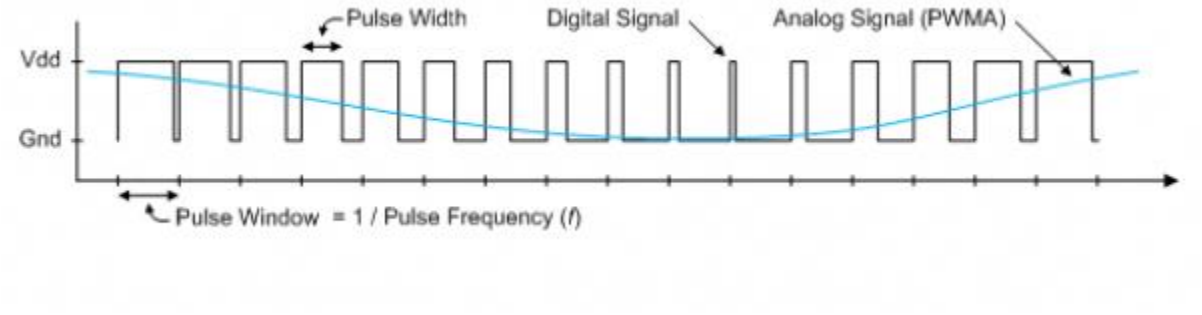


Figure 15.1.1 Simple Waveform Represented as PWM

The PWM signal must be integrated to define an analog voltage. The low-pass filter 3dB frequency should be an order of magnitude lower than the PWM frequency, so that signal energy at the PWM frequency is filtered from the signal. For example, if an audio signal must contain up to 5 KHz of frequency information, then the PWM frequency should be at least 50 KHz (and preferably even higher). In general, in terms of analog signal fidelity, the higher the PWM frequency, the better. Figure 15.1.2 shows a representation of a PWM integrator producing an output voltage by integrating the pulse train. Note the steady-state filter output signal amplitude ratio to Vdd is the same as the pulse-width duty cycle (duty cycle is defined as pulse-high time divided by pulse-window time).

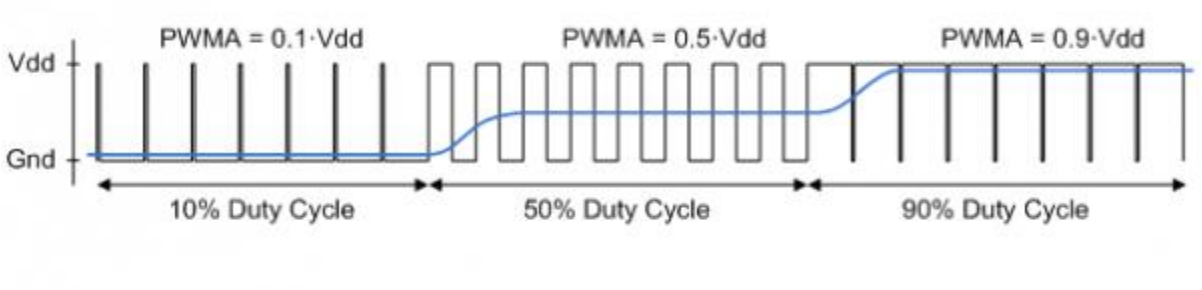


Figure 15.1.2 Representation of a PWM Integrator Producing an Output Voltage by Integrating the Pulse Train

Built-In Self-Test

A demonstration configuration is loaded into the Quad-SPI flash device on the Nexys A7 board during manufacturing. The source code and prebuilt bitstream for this design are available for download from the Digilent website. If the demo configuration is present in the flash and the Nexys A7 board is powered on in SPI mode, the demo project will allow basic hardware verification. Here is an overview of how this demo drives the different onboard components:

- The user LEDs are illuminated when the corresponding user switch is placed in the on position.
- The tri-color LEDs are controlled by some of the user buttons. Pressing BTNL, BTNC, or BTNR causes them to illuminate either red, green, or blue, respectively. Pressing BTND causes them to begin cycling through many colors. Repeatedly pressing BTND will turn the two LEDs on or off.
- Pressing BTNU will trigger a 5 second recording from the onboard PDM microphone. This recording is then immediately played back on the mono audio out port. The status of the recording and playback is displayed on the user LEDs. The recording is saved in the DDR2 memory.

- The VGA port displays feedback from the onboard microphone, temperature sensors, accelerometer, RGB LEDs, and USB Mouse.
- Connecting a mouse to the USB-HID Mouse port will allow the pointer on the VGA display to be controlled. Only mice compatible with the Boot Mouse HID interface are supported.
- The seven-segment display will display a moving snake pattern.

All Nexys A7 boards are 100% tested during the manufacturing process. If any device on the Nexys A7 board fails test or is not responding properly, it is likely that damage occurred during transport or during use. Typical damage includes stressed solder joints and contaminants in switches and buttons resulting in intermittent failures. Stressed solder joints can be repaired by reheating and reflowing solder and contaminants can be cleaned with off-the-shelf electronics cleaning products. If a board fails test within the warranty period, it will be replaced at no cost. Contact Digilent for more details.

[i] Zynq-7000 SoC and 7 Series Devices Memory Interface Solutions from Xilinx
[ii] Spansion S25FL032P_00 Datasheet
[iii] 7-Series FPGAs Configuration User Guide from Xilinx
[iv] SMSC LAN8720A Datasheet from Microchip
[v] ADXL362 Product Page from Analog Devices