Procesamiento digital de señales analógicas

Tyrone Novillo
Facultad de ingeniería
Universidad de Cuenca
Cuenca, Ecuador
tyrone.novillo@ucuenca.edu.ec

***Resumen*—This report presents the implementation of an analog-to-digital and digital-to-analog conversion system on a Basys 3 FPGA board. The process involves converting a 12-bit analog signal into a digital format using an ADC, then reconstructing the analog signal with a DAC based on an R-2R ladder network. A key aspect of the implementation is the use of a 12-bit shift register (74LS164) and three 4-bit latches (4042 CMOS) to convert serial data to parallel for DAC input. The design includes a variable-length low-pass filter to smooth the digital signal and ensure accurate representation. Results demonstrate successful reconstruction of analog signals with minimal noise, though some noise is present due to the signal's low amplitude. This work emphasizes the importance of proper synchronization in serial data transmission and the impact of filtering on signal quality.**

***Keywords*—** Analog-to-Digital Conversion, Digital-to-Analog Conversion, FPGA, Basys 3, Shift Register, Low-Pass Filter, R-2R Ladder Network

I. Introducción

En el mundo del procesamiento digital de señales, la conversión analógica-digital (ADC, por sus siglas en inglés) juega un papel crucial. Un ADC es un dispositivo que toma una señal analógica continua y la convierte en una secuencia de valores digitales discretos. Esta conversión es fundamental para que las señales del mundo real, como las de audio, video o señales de sensores, puedan ser procesadas, almacenadas y manipuladas por sistemas digitales, como microcontroladores, computadoras y FPGAs. La importancia del ADC radica en su capacidad para permitir que los sistemas digitales interactúen con el entorno físico, facilitando la aplicación de técnicas avanzadas de procesamiento y análisis de señales.

En este trabajo, se utilizará un ADC implementado en una tarjeta Basys 3 para convertir una señal analógica de 12 bits en una señal digital, que luego será reconvertida a analógica mediante un DAC basado en una red R-2R. Este proceso es esencial para visualizar la señal digitalizada en un osciloscopio y analizar su precisión y calidad. Para enviar los datos digitales al DAC, se empleará un registro de desplazamiento de 12 bits, utilizando dos circuitos integrados 74LS164 conectados en serie debido a que cada uno es de 8 bits. Este registro de desplazamiento convertirá los datos seriales en datos paralelos, los cuales serán manejados por un latch 4042 CMOS. El latch 4042, un dispositivo de 4 bits, será utilizado en tres unidades conectadas en serie para manejar los 12 bits del ADC, permitiendo así la correcta transferencia de los datos al DAC.

Este trabajo se centrará en la implementación detallada de este sistema, desde la configuración del ADC en Vivado hasta la conversión serial-paralela y la reconstrucción final de la señal analógica. Se explorarán aspectos clave como el filtro pasa bajo de longitud variable para suavizar la señal digital, la sincronización de la transmisión de datos seriales y la correcta implementación de la red R-2R para el DAC. Además, se analizarán los resultados obtenidos, evaluando la calidad de la señal reconstruida y la influencia de factores como el ruido y la tasa de muestreo. Este estudio proporciona una comprensión integral del proceso de conversión analógico-digital y digital-analógico, destacando su relevancia y aplicaciones prácticas en el campo del procesamiento de señales.

II. Desarrollo

Se requiere convertir una señal anaógica en digital mediante el conversor ADC implementado en la Basys 3 de 12 bits. Dichas señal será convertida posteriormente en anaógica para ser mostrada en un osciloscopio. Esta conversión se la hará mediante un DAC (red R-2R). Las entradas digitales de este conversor serán enviadas serialmente por la Basys y estos registros de 12 bits serán convertidos serialmente a paralelamente mediante un registro de desplazamiento de 12 bits y un latch para enviar los datos paralelamente. La Figura 1 muestra un diagrama del circuito a implementar. Para la conversión de datos paralelos a seriales, se hará uso del registro de desplazamiento 74LS164 y para enviar los datos hacia el DAC (red R-2R), se utilizará el latch 4042 CMOS. Cabe mencionar que se hará uso de 2 circuitos integrados 74LS164 conectados en serie debido a que dicho integrado es de 8 bits y la salida del ADC de 12. Además, para enviar los datos paralelamente al DAC, se requieren 3 latch 4042 debido a que cada uno es de 4 bits.

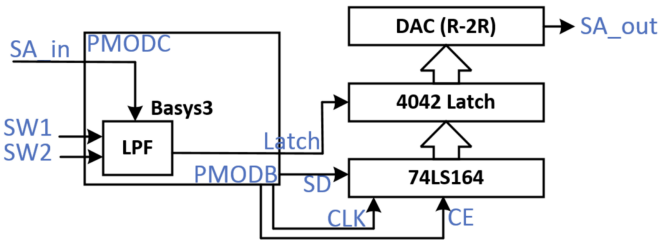

Figura 1: Esquema del hardware

II-A. ADC

Para iniciar, se explicaá como tomar la señal analógica y convertirla a digital mediante el ADC implementado en la FPGA. Para esto se debe configurar desde el setup wizard que tiene integrado Vivado.

Para configurar el XADC, se deben seguir los siguientes pasos: Primero dirigirse al menú de *LogicCORE IP* (Propiedad intelectual) proporcionado por Vivado, como muestra la Figura 2

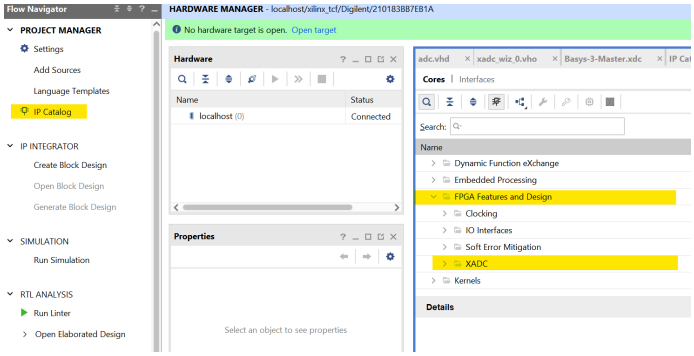

Figura 2: Menú del LogicCORE IP

Una vez en el wizard, se deben colocar los siguientes datos:

- **Interface options:** DRP
- **Timing mode:** Event mode
- **Startup Channel Selection:** Single Channel
- **DCLK Frequency:** 100MHz
- **Control/Status Port:** reset in
- **Event mode Trigger:** convst_in
- **Sequencer mode:** off
- **Todas las alarmas apagadas**
- **Selected Channel:** VAUXP5 VAUXN5
- **Bipolar:** off

Una vez configurados estos parámetros, el componente que proporciona Vivado es uno como de la Figura 3

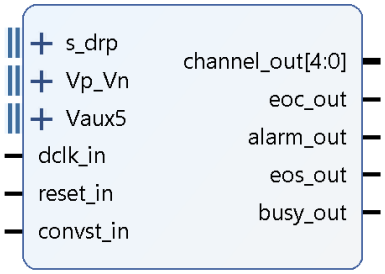

Figura 3: Componente ADC

Este documento describe un módulo llamado `adc`, diseñado para ser implementado en una tarjeta Basys 3, utilizando un conversor analógico-digital XADC (Xilinx Analog-to-Digital Converter). El diseño se estructura en una entidad y una arquitectura, donde se declaran y configuran las señales necesarias para el funcionamiento del XADC.

La entidad `adc` define las entradas y salidas del módulo. Tiene las siguientes señales:

- `clk`: Señal de entrada de reloj (`STD_LOGIC`).
- `sw`: Señal de entrada de conmutador (`STD_LOGIC`).
- `x`: Señal de salida del conversor ADC(`STD_LOGIC_VECTOR (15 downto 0)`).
- `JA`: Señal de entrada que recibe datos desde el conector JA de la tarjeta (`STD_LOGIC_VECTOR (7 downto 0)`).

La Figura 4 muestra el muestreo de la señal anaógica sobre la variable llamada x. Como se puede observar, al tener 12 bits, la resolución de la señal digital es considerablemente buena.

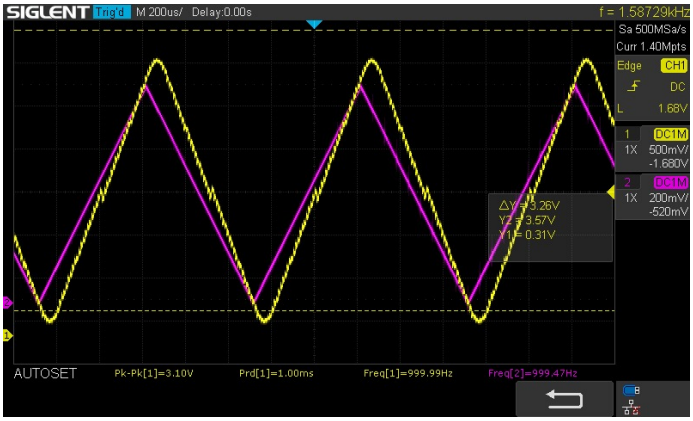

Figura 4: Salida del ADC

Dentro de la arquitectura `Behavioral`, se declara un componente llamado `xadc_wiz_0`, que representa el módulo XADC proporcionado por Xilinx. Básicamente, la salida del conversor ADC se llama en este caso `x` y para este caso, se lo hizo a una frecuencia de muestreo de 100KHz.

II-B. Filtro pasa bajo LPF

A continuación, se explicará el procedimiento para generar el filtro pasa bajos digital de longitud variable para instanciarlo posteriormente. La entidad `final` define los puertos de entrada y salida del sistema, incluyendo una señal de reloj (`clk_in`), una señal de fin de conversión (`eoc`) proporcionado por el ADC cuya frecuencia es equivalente a la frecuencia de muestreo predefinida anteriormente. Este componente también tiene una entrada de datos de 12 bits (`data_in`) que corresponde a la salida de datos del ADC, es decir los datos de 12 bits. También requiere una entrada de un selector de longitud del filtro de 2 bits (`len_lpf`) y finalmente tendrá una salida de datos filtrados de 12 bits (`data_out`).

Dentro de la arquitectura , se declara un tipo de dato (`tipo_arreglo_muestras`) que define un arreglo de registros de 12 bits para almacenar hasta 8 muestras de datos. Se definen varias señales: `suma`, para almacenar la suma de las muestras; `promedio_int`, para almacenar el promedio calculado como un entero; `longitud_filtro`, para almacenar la longitud del filtro seleccionado; y `registro_muestras`, que es el registro de desplazamiento que contiene las muestras de datos.

El proceso principal se activa en cada flanco ascendente de `clk_in` de 100MHz. Cuando `eoc` está en alto (cuando se termina una conversión), se realiza una operación de registro de desplazamiento, donde las muestras previas se desplazan hacia arriba y la nueva muestra (`data_in`) se almacena en la primera posición del registro. Luego, según el valor de `len_lpf`, se calcula la suma de las muestras:

- Si `len_lpf` es ”01”, se calcula la suma de las primeras dos muestras y se ajusta `longitud_filtro` a 2.
- Si `len_lpf` es ”10”, se calcula la suma de las primeras cuatro muestras y se ajusta `longitud_filtro` a 4.
- Si `len_lpf` es ”11”, se calcula la suma de las ocho muestras y se ajusta `longitud_filtro` a 8.
- En cualquier otro caso, se toma solo la muestra actual (`data_in`) y se ajusta `longitud_filtro` a 1.

Finalmente, se calcula el promedio dividiendo la suma por la longitud del filtro y se convierte el resultado a un vector de lógica estándar de 12 bits, que se asigna a `data_out`. Este proceso asegura que la salida del filtro es una versión suavizada de las entradas, promediada según la longitud seleccionada del filtro. Las Figuras 5, 6, 7 muestran las diferencias de tomar 2, 4 y 8 muestras para promediarlas. Como se puede observar, el efecto del filtro es suavizar

cuando se promedian el menor número de muestras. Esto tiene sentido debido a que cuando se toman muchas muestras lejanas, el valor de la muestra tiende a alejarse al valor real de medición.

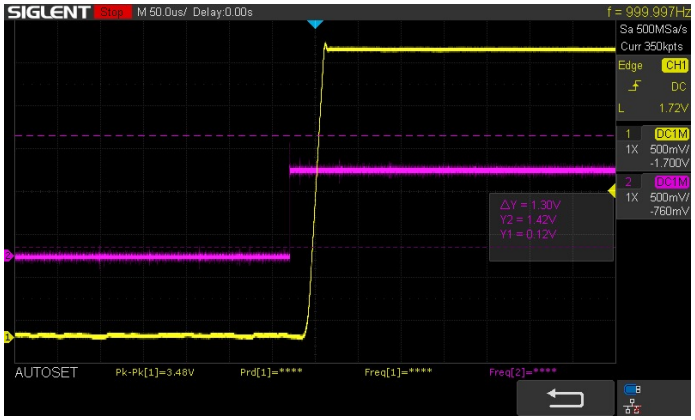

Figura 5: Señal de ADC aplicada el filtro con 2 bits de longitud

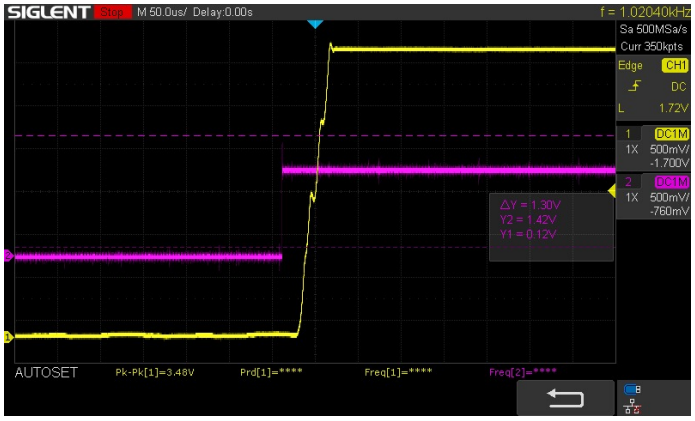

Figura 6: Señal de ADC aplicada el filtro con 4 bits de longitud

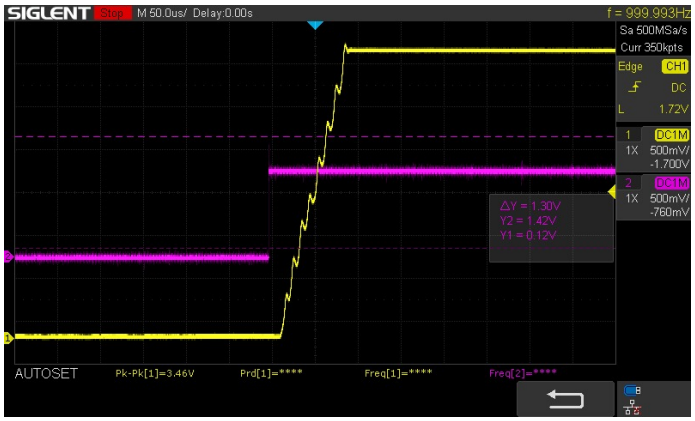

Figura 7: Señal de ADC aplicada el filtro con 8 bits de longitud

II-C. Envío de datos seriales

Para enviar serialmente los datos obtenidos por el filtro, es de mucha importancia tomar en cuenta la frecuencia de muestreo del ADC. Para este caso, la frecuencia de muestreo es de 100KHz, por lo que antes de que se genere un pulso de esta señal de muestreo, se deben haber enviado los 12 bits. Así que esta frecuencia debe ser mayor a la de muestreo pero solamente debe tener 12 pulsos para enviar los 12 datos dentro del 74LS164. Para aquello se define el siguiente proceso para generar una señal más rápida con 12 pulsos a partir de la bandera de conversión del ADC.

Para ello, primero se crea un proceso que generará una señal de reloj para enviar datos, sincronizado con una señal de fin de conversión (`eoc_out`) de 100 kHz. El objetivo es generar una señal de 12 pulsos de 2 MHz una vez que `eoc_out` se pone en estado lógico alto.

El proceso principal se activa en cada flanco ascendente de la señal de reloj (`clk`). Dentro de este proceso, se utilizan dos contadores: `contador_2M` para contar los ciclos necesarios para generar una señal de 2 MHz y `contador_12` para contar los 12 pulsos de esta señal.

Cuando `eoc_out` está en alto, ambos contadores (`contador_2M` y `contador_12`) se reinician a 0. El proceso continúa de la siguiente manera:

- Si `contador_12` es menor que 12, `contador_2M` se incrementa en cada ciclo de reloj.
- Cuando `contador_2M` alcanza 24, se reinicia a 0 y la señal `clk_serial` cambia su estado (alternando entre alto y bajo).
- Si `clk_serial` está en alto después de cambiar de estado, `contador_12` se incrementa en 1.
- Si `contador_12` alcanza 12, `clk_serial` se pone en estado bajo y deja de alternar.

Este proceso crea la señal de reloj de 12 pulsos con una ventana para evitar la superposición de pulsos y por lo tanto, pérdida de bits de muestras del filtro. La Figura 8 muestra en morado la señal con la frecuencia de muestreo, y en amarillo, los 12 pulsos que se genera cuando se tiene una muestra lista.

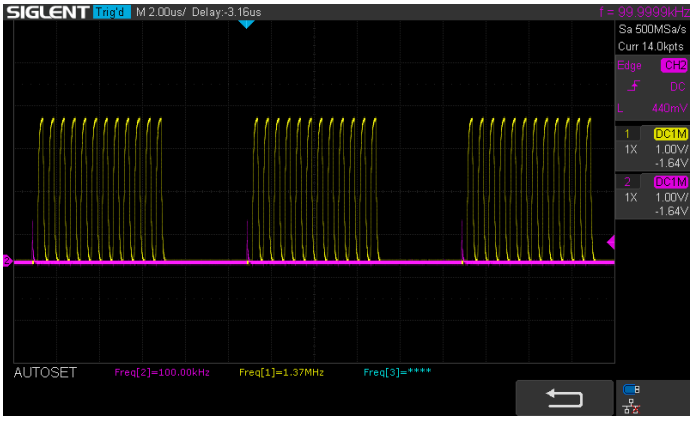

Figura 8: Reloj para enviar 12 datos seriales tras haber generado un valor por el ADC

Cabe mencionar que la señal de eoc_out no está proporcionando 3.3V sino más bien 1V. Se ha intentado algunos métodos para hacer que el voltaje del estado lógico de esta señal sea mayor debido a que podría estar afectando a la resolución de las señal de salida final.

Una vez generada la señal para enviar los 12 datos seriales por el pin de salida de la Basys 3, es necesario entender como será la ruta que tomarán los bits desde la salida serial hasta la red DAC que se definirá posteriormente. En este caso, como se mencionó, se utilizará el registro de desplazamiento 74LS174 conectado en cascada (debido a que es de 8 bits) con otro del mismo tipo. Este registro lo que hace es desplazar los datos para ordenarlos. Para ese caso, se ha implementado de tal manera que los bits menos significativos salgan primero serialmente y al final, el bit más significativos se posicionen en el primer registro del primer 74LS174. Para enviar estos datos

ordenados previamente de forma paralela, se hará uso del latch que simplemente lo que hace es transmitir datos paralelos almacenados hacia salidas paralelas. La señal de reloj que define este latch será el mismo eoc_out, es decir, cada vez que se genere una nueva muestra, el valor almacenado en los 74LS164 se enviarán al DAC. La Figura 9 muestra una simulación del uso de estos circuitos integrados previo a la implementación real.

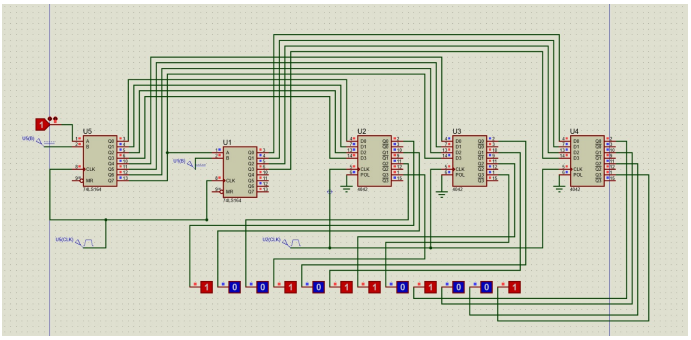

Figura 9: Simulación en Proteus del registro de desplazamiento.

Para enviar los datos seriales por el pin de la Basys 3 se utiliza un proceso sincronizado con la señal de reloj serial definida previamente. El objetivo es enviar los bits de un vector (`y`) uno por uno, comenzando por el bit menos significativo (LSB). Es importante mencionar que el circuito simulado en Proteus presentado está diseñado para cuando se envía primero el bit menos significativo. Si se enviara primero el más significativo, la conexión de la entrada del DAC con las salidas del latch 4042 se invierte.

Dentro del proceso, se utiliza una variable `count_serial` para llevar un conteo de los bits enviados. El funcionamiento del proceso es el siguiente:

- Si `count_serial` es igual a 11, significa que se han enviado todos los bits del vector (`y`), por lo que `count_serial` se reinicia a 0 y la señal de datos seriales (`SD`) se pone en estado bajo (`'0'`).
- Si `count_serial` es menor que 11, se asigna el valor del bit correspondiente de `y` a `SD`, comenzando por el bit menos significativo.
- Luego, `count_serial` se incrementa en 1 para preparar el envío del siguiente bit en el siguiente ciclo de reloj.

II-D. Conversor DAC

Una vez que se tienen listos los valores en binario de 12 bits de forma paralela, se deben ingresar estos datos en el conversor DAC. En este caso, se utilizará una red R-2R, como se muestra en la Figura 10. Siguiendo las recomendaciones de docentes de la universidad, se utilizaron resistencias de 10KΩ. Además, para reducir el efecto del ruido, se procede a colocar al final de la red un seguidor de tensión del integrado M358.

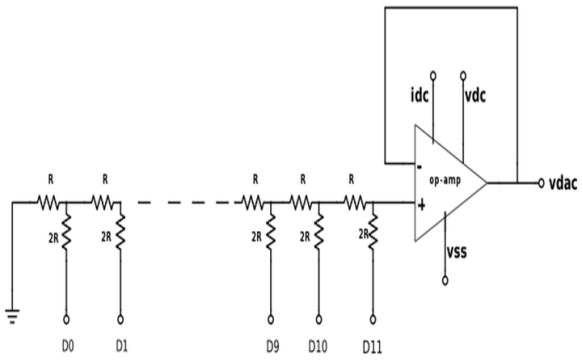

Figura 10: Diagrama del circuito de la red R-2R.

II-E. Resultados

Como se aprecia en las figuras 11, 12, y 13, la señal reconstruida presenta ruido, sin embargo, se ajusta a la señal analógica original original. Adicionalmente, al variar la frecuencia de la señal analógica, se puede demostrar que, al superar la tasa de Nyquist, la señal resultante si que se ajusta a la señal generada, es decir, se reconstruye la señal mediante sus muestras. Esto corresponde al teorema del muestreo. Cuando la frecuencia del generador aumenta, la señal resultante se atenua.

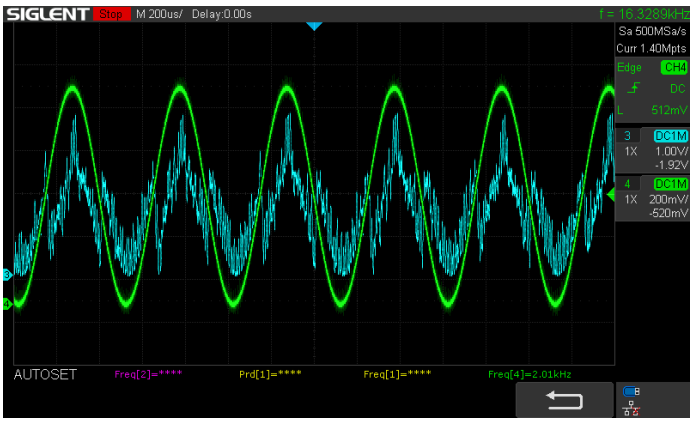

Figura 11: Salida del conversor DAC para una señal anaógica senoidal.

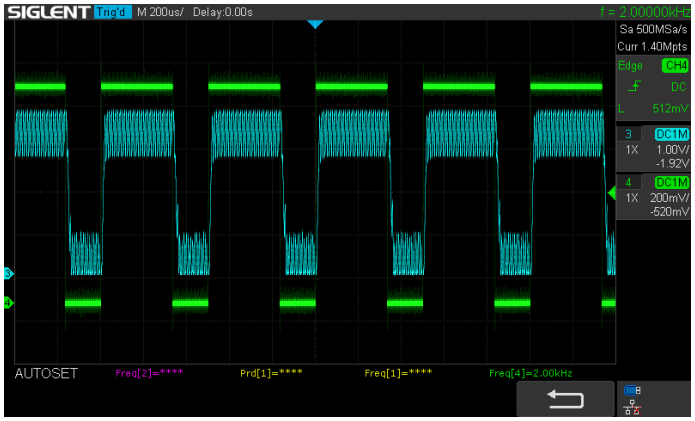

Figura 12: Salida del conversor DAC para una señal anaógica de pulsos cuadrados.

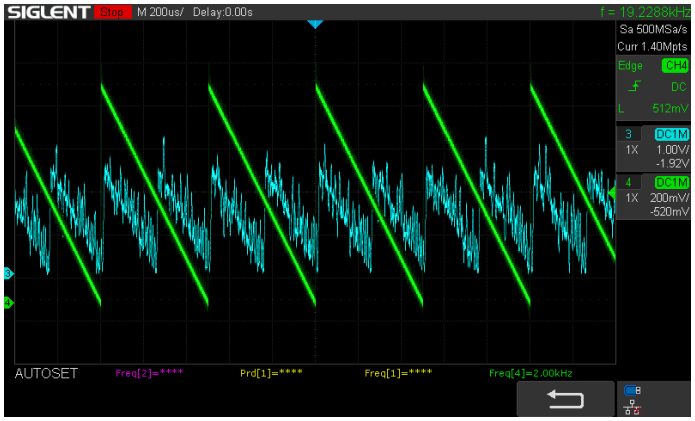

Figura 13: Salida del conversor DAC para una señal anaógica triangular.

III. Conclusiones

A pesar de seguir cuidadosamente las recomendaciones brindadas por los datasheet, ajustar un tiempo de ventana en el reloj para enviar datos seriales o incluir un seguidor de tensión, la señal reconstruida no es exactamente igual a la señal introducida al ADC de la Basys. Cabe mencionar que introducir el amplificador como seguidor de tensión si que atenuó la señal de ruido en la senñal final sin embargo no la eliminó por completo. Uno de los principales problemas en el diseño de este tipo de convertidores serial a paralelo con la Basys 3 es que las señales de reloj generadas por la FPGA no tienen amplitud de 5V, sino más bien de 3.3V (a diferencia de la señal eoc_out que era de 1V), necesario para entradas de integrados de tecnología CMOS. Adicionalmente, se pudo verificar el teorema de Nyquist para la recontrucción de la señal a partir de sus muestras siempre cuando la frecuencia de muestreo sea al menos dos veces mayor a la frecuencia de la señal a reconstruir.

IV. Anexos

IV-A. Archivo final.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
USE IEEE.std_logic_unsigned.ALL;

entity final is
Port (clk_in : in std_logic;
-- Señal de reloj
eoc: in std_logic;
-- Señal de fin de conversión
data_in: std_logic_vector(11 downto 0);
-- Entrada de datos (12 bits)
len_lpf: in std_logic_vector(1 downto 0);
-- Selector de longitud del filtro
data_out: out std_logic_vector(11 downto 0));
-- Salida de datos filtrados (12 bits)
end final;

architecture Behavioral of final is
-- Tipo de dato para el registro de desplazamiento que
type tipo_arreglo_muestras is array(0 to 7) of STD_LOG
signal suma: STD_LOGIC_VECTOR(15 downto 0);
-- Señal para almacenar la suma de las muestras
signal promedio_int: integer := 0;
-- Señal para almacenar el promedio calculado
signal longitud_filtro : integer := 0;
-- Señal para almacenar la longitud del filtro
signal registro_muestras: tipo_arreglo_muestras;
-- Registro de desplazamiento para almacenar muestras

begin
process(clk_in)
begin
if rising_edge(clk_in) then
if eoc = '1' then
-- Operación del registro de desplazamiento
for i in 7 downto 1 loop
registro_muestras(i) <= registro_muestras(i-1);
end loop;
registro_muestras(0) <= data_in; -- Cargar nueva muest
end if;

-- Calcular la suma de las muestras basado en la longi
case len_lpf is
when "01" => -- Longitud 2
suma <= ("0000" & registro_muestras(0)) + ("0000" & re
longitud_filtro <= 2;
when "10" => -- Longitud 4
suma <= ("0000" & registro_muestras(0)) + ("0000" & re
longitud_filtro <= 4;
when "11" => -- Longitud 8
suma <= ("0000" & registro_muestras(0)) + ("0000" & re
("0000" & registro_muestras(4)) + ("0000" & registro_m
longitud_filtro <= 8;
when others => -- Por defecto a longitud 1 (sin promed
suma <= "0000" & data_in;
longitud_filtro <= 1;
end case;

-- Calcular el promedio de las muestras
promedio_int <= TO_INTEGER(unsigned(suma)) / longitud_
-- División por el número de muestras
data_out <= std_logic_vector(to_unsigned(promedio_int,
-- Salida de la muestra promediada
end if;
end process;
end Behavioral;
```

IV-B. Archivo main.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```vhdl
entity main is
Port ( clk    : in STD_LOGIC;
btn    : in STD_LOGIC;
JA     : in STD_LOGIC_VECTOR (7 downto 0);
sw     : in STD_LOGIC_VECTOR (1 downto 0);
SD     : out std_logic;
CLK_SD : out std_logic;
CLEAR  : out std_logic;
LATCH  : out std_logic
);
end main;

architecture Behavioral of main is
COMPONENT xadc_wiz_0
PORT (
di_in : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
daddr_in : IN STD_LOGIC_VECTOR(6 DOWNTO 0);
den_in : IN STD_LOGIC;
dwe_in : IN STD_LOGIC;
drdy_out : OUT STD_LOGIC;
do_out : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
dclk_in : IN STD_LOGIC;
reset_in : IN STD_LOGIC;
convst_in : IN STD_LOGIC;
vp_in : IN STD_LOGIC;
vn_in : IN STD_LOGIC;
vauxp5 : IN STD_LOGIC;
vauxn5 : IN STD_LOGIC;
channel_out : OUT STD_LOGIC_VECTOR(4 DOWNTO 0);
eoc_out : OUT STD_LOGIC;
alarm_out : OUT STD_LOGIC;
eos_out : OUT STD_LOGIC;
busy_out : OUT STD_LOGIC
);
END COMPONENT;

component final is
Port(
clk_in : in std_logic;
eoc: in std_logic;
data_in: std_logic_vector(11 downto 0);
len_lpf: in std_logic_vector(1 downto 0);
data_out: out std_logic_vector(11 downto 0));
end component;

signal channel_out  : std_logic_vector(4 downto 0);
signal daddr_in : std_logic_vector(6 downto 0);
signal eoc_out : std_logic;
signal do_out : std_logic_vector(15 downto 0);
signal anal_p, anal_n, convst, drdy: std_logic;
signal count : integer range 0 to 999;
signal x, y : std_logic_vector(11 downto 0);


--Senales para el envio de datos seriales

signal clk_serial: STD_LOGIC;
signal contador_2M : integer range 0 to 63;
signal contador_12: integer range 0 to 12;

begin

daddr_in <= "00" & channel_out;
anal_p <= JA(4);
anal_n <= JA(0);

filtro_variable : final
PORT MAP(clk_in   => clk,
eoc      => eoc_out,
data_in  => x,
len_lpf  => sw,
data_out => y);

your_adc : xadc_wiz_0
PORT MAP (
di_in => "0000000000000000",
daddr_in => daddr_in,
den_in => eoc_out,
dwe_in => '0',
drdy_out => open,
do_out => do_out,
dclk_in => clk,
reset_in => btn,
convst_in => convst,
vp_in => '0',
vn_in => '0',
vauxp5 => anal_p,
vauxn5 => anal_n,
channel_out => channel_out,
eoc_out => eoc_out,
alarm_out => open,
eos_out => open,
busy_out => open
);

x  <= do_out(15 downto 4);


--Senal de reloj de 100KHz (Frecuencia de muestreo)
process(clk)
begin
if rising_edge(clk) then
count <= count + 1;
convst <= '0';
if count = 999 then
count <= 0;
convst <= '1';
end if;
end if;
end process;

--Generar senal de reloj para enviar datos
process(clk)
begin
if rising_edge(clk) then
if eoc_out = '1' then
contador_2M <= 0;
contador_12 <= 0;
end if;

if contador_12 < 12 then
contador_2M <= contador_2M + 1;
if contador_2M = 24 then
contador_2M <= 0;
clk_serial <= not clk_serial;
if clk_serial = '1' then
contador_12 <= contador_12 + 1;
end if;
end if;
else
clk_serial <= '0';
end if;
end if;
```

```
end process;

--Enviar datos serialmente
process (clk_serial) is
variable count_serial : integer := 0;
begin
if rising_edge(clk_serial) then
if count_serial = 11 then
count_serial := 0;
SD <= '0';
else
SD <= y(count_serial);
count_serial := count_serial + 1;
end if;
end if;
end process;


CLK_SD  <= clk_serial;
LATCH   <= eoc_out;
CLEAR   <= '1';

end Behavioral;
```

IV-C. Archivo de Constraints Basys-3-Master.xdc

```
# Clock signal
set_property -dict { PACKAGE_PIN W5
↪  IOSTANDARD LVCMOS33 } [get_ports clk]
create_clock -add -name sys_clk_pin -period
↪  10.
set_property -dict { PACKAGE_PIN V17
↪  IOSTANDARD LVCMOS33 } [get_ports {sw[0]}]
set_property -dict { PACKAGE_PIN V16
↪  IOSTANDARD LVCMOS33 } [get_ports {sw[1]}]

#Buttons
set_property -dict { PACKAGE_PIN U18
↪  IOSTANDARD LVCMOS33 } [get_ports btn]

#Pmod Header JA
set_property -dict { PACKAGE_PIN J1
↪  IOSTANDARD LVCMOS33 } [get_ports
↪  {JA[0]}];#Sch name = JA1
set_property -dict { PACKAGE_PIN L2
↪  IOSTANDARD LVCMOS33 } [get_ports
↪  {JA[1]}];#Sch name = JA2
set_property -dict { PACKAGE_PIN J2
↪  IOSTANDARD LVCMOS33 } [get_ports
↪  {JA[2]}];#Sch name = JA3
set_property -dict { PACKAGE_PIN G2
↪  IOSTANDARD LVCMOS33 } [get_ports
↪  {JA[3]}];#Sch name = JA4
set_property -dict { PACKAGE_PIN H1
↪  IOSTANDARD LVCMOS33 } [get_ports
↪  {JA[4]}];#Sch name = JA7
set_property -dict { PACKAGE_PIN K2
↪  IOSTANDARD LVCMOS33 } [get_ports
↪  {JA[5]}];#Sch name = JA8
set_property -dict { PACKAGE_PIN H2
↪  IOSTANDARD LVCMOS33 } [get_ports
↪  {JA[6]}];#Sch name = JA9
set_property -dict { PACKAGE_PIN G3
↪  IOSTANDARD LVCMOS33 } [get_ports
↪  {JA[7]}];#Sch name = JA10

Pmod Header JB
set_property -dict { PACKAGE_PIN A14
↪  IOSTANDARD LVCMOS33 } [get_ports
↪  {CLK_SD}];#Sch name = JB1
set_property -dict { PACKAGE_PIN A16
↪  IOSTANDARD LVCMOS33 } [get_ports
↪  {LATCH}];#Sch name = JB2
set_property -dict { PACKAGE_PIN B15
↪  IOSTANDARD LVCMOS33 } [get_ports
↪  {CLEAR}];#Sch name = JB3
set_property -dict { PACKAGE_PIN B16
↪  IOSTANDARD LVCMOS33 } [get_ports
↪  {SD}];#Sch name = JB4
#set_property -dict { PACKAGE_PIN A15
↪  IOSTANDARD LVCMOS33 } [get_ports
↪  {JB[4]}];#Sch name = JB7
#set_property -dict { PACKAGE_PIN A17
↪  IOSTANDARD LVCMOS33 } [get_ports
↪  {JB[5]}];#Sch name = JB8
#set_property -dict { PACKAGE_PIN C15
↪  IOSTANDARD LVCMOS33 } [get_ports
↪  {JB[6]}];#Sch name = JB9
#set_property -dict { PACKAGE_PIN C16
↪  IOSTANDARD LVCMOS33 } [get_ports
↪  {JB[7]}];#Sch name = JB10

## Configuration options, can be used for all
↪  designs
set_property CONFIG_VOLTAGE 3.3
↪  [current_design]
set_property CFGBVS VCCO [current_design]

## SPI configuration mode options for QSPI
↪  boot, can be used for all designs
set_property BITSTREAM.GENERAL.COMPRESS TRUE
↪  [current_design]
set_property BITSTREAM.CONFIG.CONFIGRATE 33
↪  [current_design]
set_property CONFIG_MODE SPIx4
↪  [current_design]
```

IV-D. Enlace al código de Arduino en GitHub

https://github.com/thyron001/VHDL/tree/main/Trabajo%20final