Oscilador senoidal controlado numéricamente (NCO senoidal)

Guillermo F. Caporaletti

gcaporaletti@fi.uba.ar

Junio de 2023

Carrera de Especialización en Sistemas Embebidos
Curso de Circuitos Lógicos Programables

Índice

1. Introducción 3
   1.1. Descripción general 3
   1.2. Ecuaciones básicas 4
   1.3. Variando la frecuencia 4
   1.4. Resolución y rango de frecuencia 4
   1.5. Obtención de I correspondiente a una frecuencia 5
2. Implementación 6
   2.1. Especificaciones y componentes 6
   2.2. Conversor Fase-Amplitud 7
   2.3. Acumulador de Fase 8
   2.4. Integración de componentes y simulación 9
   2.5. Síntesis e implementación para prueba en Arty Z7-10 10
   2.6. Utilización de IP cores para prueba remota 12
3. Conclusiones 15
4. Documentos y referencias 16
5. Definiciones, acrónimos y abreviaturas 16

1. Introducción

El presente trabajo fue realizado en el curso de Circuitos Lógicos Programables de la Carrera de Especialización en Sistemas Embebidos, Facultad de Ingeniería, UBA. Consiste en la implementación de un oscilador senoidal controlado numéricamente en un FPGA. NCO senoidal, según sus siglas en inglés (por *numerically controlled oscillator*).

1.1. Descripción general

Un NCO senoidal implementa una señal senoidal a partir de una base de tiempo y componentes digitales. Frente a los circuitos analógicos, uno de los objetivos de esta forma de implementación es evitar la necesidad de calibración debido a los cambios en los componentes producidos con el paso del tiempo. Por otra parte, puede permitir un amplio rango de frecuencias sin sumar excesiva complejidad. Además, facilita el control remoto del sistema.

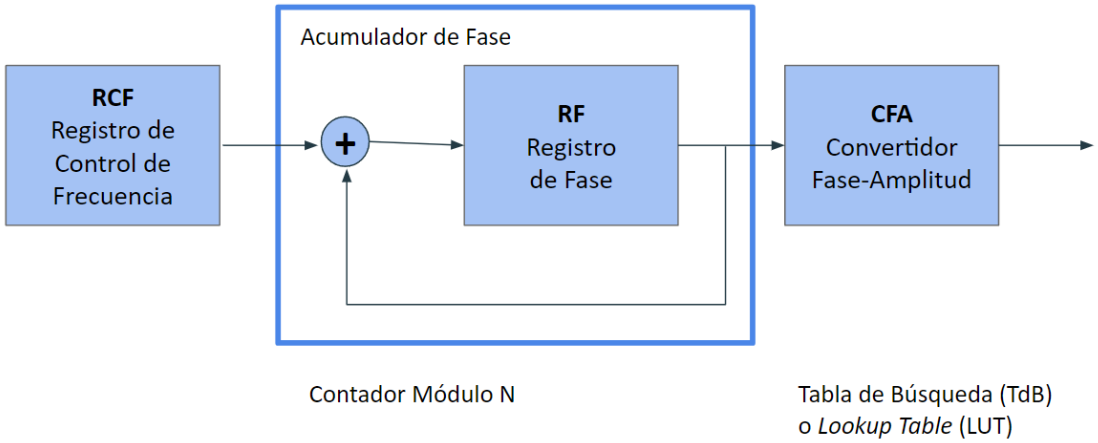

Figura 1. *Diagrama básico de un NCO.*

En la Figura 1 se parecían las partes componentes del sistema. El Convertidor Fase-Amplitud (CFA) es una tabla de búsqueda que almacena en memoria los valores de cada muestra de la señal. Si la discretización en el tiempo es de N=10 bits por período, la cantidad de muestras será:

$$2^{10}\ muestras = 1024\ muestras \qquad (1)$$

En principio, esto implicará 1024 direcciones de memoria, aunque también puede reducirse a un cuarto de esa cantidad si aprovecháramos las simetrías de la señal senoidal. En nuestro caso utilizaremos 1024 direcciones para simplificar el diseño. Esta discretización en el tiempo de 10 bits sería razonable para un DAC con 10 bits de discretización en la amplitud. En nuestro caso utilizaremos una discretización de amplitud de 12 bits.

El Registro de Fase (RF) con la realimentación constituye un Acumulador de Fase, que es de hecho un contador. Este contador debe ser de módulo N para recorrer todas las direcciones de la tabla de búsqueda CFA. Cuando el contador desborda, se vuelve a la fase 0 o valor inicial de la tabla. El Registro de Control de Frecuencia (RCF) puede ser directamente el reloj del sistema.

Este sistema va a ir recorriendo todas las direcciones del CFA de forma periódica. Así, la señal almacenada en el CFA se irá mostrando a su salida.

1.2. Ecuaciones básicas

Partiendo de la Figura 1, puntualizamos las siguientes ecuaciones:

N	Bits del contador
2^N	Cantidad de muestras por período
$T_{Reloj} = \frac{1}{F_{Reloj}}$	Período del reloj o base de tiempo del sistema [seg]
$T_0 = T_{Reloj} * 2^N$	Período de la señal de salida [seg]
$F_0 = \frac{F_{Reloj}}{2^N}$	Frecuencia de la señal de salida [Hz]

Según las ecuaciones (2), la frecuencia de salida es la frecuencia del reloj del sistema dividido la cantidad de muestras. Esta sería la máxima frecuencia posible con esa cantidad de muestras, 1024 en nuestro caso. Para frecuencias más altas, debemos bajar la cantidad de muestras.

En nuestro caso $F_{Reloj} = 125\ \text{MHz}$. Para el caso de N=10, la frecuencia de la señal resultaría:

$$F_0 = \frac{125\ \text{MHz}}{1024} = 122{,}1\ \text{kHz} \qquad (3)$$

1.3. Variando la frecuencia

Para variar la frecuencia introducimos un parámetro I para variar el incremento del contador. Con I=2, la cantidad de muestras de la señal será 512 en lugar de 1024. De este modo, se degrada la señal aunque logramos aumentar su frecuencia dos veces. Introduciendo este nuevo parámetro, las ecuaciones quedan del siguiente modo:

$T_0 = \frac{T_{Reloj} * 2^N}{I}$	Período de la señal de salida [seg]
$F_0 = \frac{F_{Reloj} * I}{2^N}$	Frecuencia de la señal de salida [Hz]

Nótese que la frecuencia de la señal es proporcional a I. Esto es una gran ventaja a la hora de configurar la frecuencia deseada. Por otra parte, I puede ser un número fraccional, lo que permite no sólo subir sino también bajar la frecuencia.

1.4. Resolución y rango de frecuencia

La frecuencia máxima depende de cuánto se pueda degradar la señal achicando la cantidad de muestras. Si tomamos un valor máximo de $I_{Máximo} = 17$, nuestra señal se degradará a 60 muestras y la frecuencia máxima será:

$$F_{0\,Máxima} = \frac{125\ \text{MHz} * I_{Máximo}}{1024} = 2{,}075\ \text{MHz} \qquad (5)$$

Si tomamos a I con valores enteros, nuestra frecuencia mínima y la resolución en frecuencia será directamente los 122,1 kHz calculados en la ecuación (3). Si en cambio permitimos valores de I con fracciones, podemos obtener valor de frecuencia más bajos y con una mejor resolución.

Si por ejemplo agregamos una cantidad R=20 de bits para la parte fraccional, la frecuencia mínima y resolución quedará establecida en:

$$F_{MIN} = \frac{F_{Reloj} * I_{MIN}}{2^N} = \frac{F_{Reloj}}{2^N * 2^R} = \frac{F_{Reloj}}{2^{N+R}} = 0,1164\ Hz$$

De esta forma, estaríamos logrando un generador con una frecuencia que va desde 0,1 Hz hasta 2 MHz. La resolución de 0,1 Hz en 2 MHz es 1/20 de parte por millón. Típicamente, la estabilidad de un cristal de cuarzo utilizado para la frecuencia de reloj es de 50 ppm[4]. Desde este punto de vista, esta resolución es irreal para estas frecuencias altas. Sí tiene sentido para las frecuencias bajas y medias del orden de los kilohertz.

1.5. Obtención de I correspondiente a una frecuencia

Si necesitamos obtener el valor de I, y más precisamente su formato binario, debemos despejar I de la ecuación (4). Si por ejemplo buscamos una frecuencia de 10 kHz, debemos hacer:

$$F_0 = 10\ \text{kHz} = \frac{125\ \text{MHz}}{1024} * I = 122,1\ \text{kHz} * I \tag{7}$$

$$I = \frac{10\ \text{kHz} * 1024}{125\ \text{MHz}} = 0,08192 \tag{8}$$

Este valor, expresado en binario, equivale a 0,00010100111110001011. Si recalculamos el valor de frecuencia para este número binario, la misma resulta $F_0 = 9999,96\ \text{Hz}$. Esta aproximación es más precisa que la estabilidad típica de un cristal de cuarzo.

2. Implementación

2.1. Especificaciones y componentes

En la Tabla 1 resumimos los valores elegidos para nuestro prototipo. En la Figura 2 se muestra un diagrama de bloques del sistema diseñado para implementar.

Tabla 1. Características del NCO senoidal a implementar

Parámetro	Descripción	Magnitud	Unidad
$F_{0,Máxima}$	Frecuencia máxima nominal	2.000.000	Hz
ΔF	Resolución nominal en frecuencia	0,25	Hz
N	Bits de muestreo	10	b
Q	Cuantización del DAC	12	b
M	Bits de la parte entera de I	5	b
R	Bits de la parte fraccional de I	20	b
M+R	Bits totales de I y J	25	b
C = N+R	Bits del Acumulador de Fase	30	b

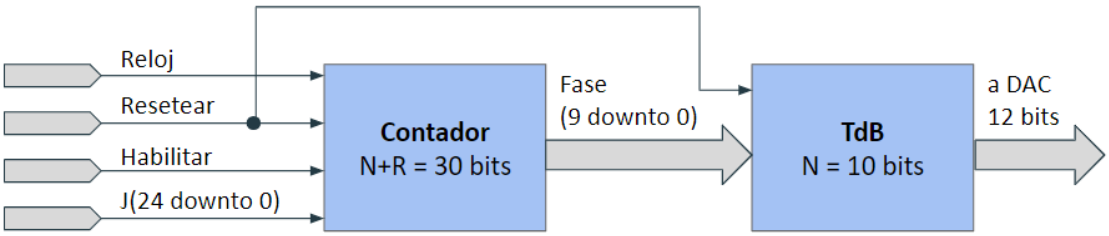

Figura 2. *Diagrama en bloques del oscilador a implementar.*

El CFA está constituido por una TdB de 10 bits de entradas (1024 direcciones) y 12 bits de salida. El valor de 12 bits es un valor típico de DAC's y ADC's. Su funcionamiento será asincrónico y compartirá el reset con el contador.

El Acumulador de Fase será resuelto con un contador que tiene una particularidad: los pasos de la cuenta no son sólo de a 1 sino que permite introducirse como parámetro. J es un vector de 25 bits que representa un número entero sin signo. Es otra forma de escribir a I, que tiene M=5 para su parte entera y otros R=20 para su parte fraccional. Como salida, sólo tomamos los 10 bits más significativos del contador.

En la Figura 3 mostramos las correspondencia de los bits del incremento J (de 25 bits), el contador del Acumulador de Fase (30 bits) y la fase de salida al CFA (10 bits). Como se aprecia, los bits menos significativos del incremento y del contador coinciden. Por otra parte, los bits más significativos del contador coinciden con los más significativos de la fase de salida, que va al CFA. Los 5 bits que se superponen entre el incremento y la fase corresponden a la parte entera de I. Es decir, los valores que implican que la señal de salida tenga menos muestras para lograr aumentar la frecuencia. El resto de los bits del incremento (desde 0 a 19) corresponden a la parte fraccional del incremento I. La combinación de los bits de las partes entera y fraccional de I constituyen el vector de bits J, que es un número entero.

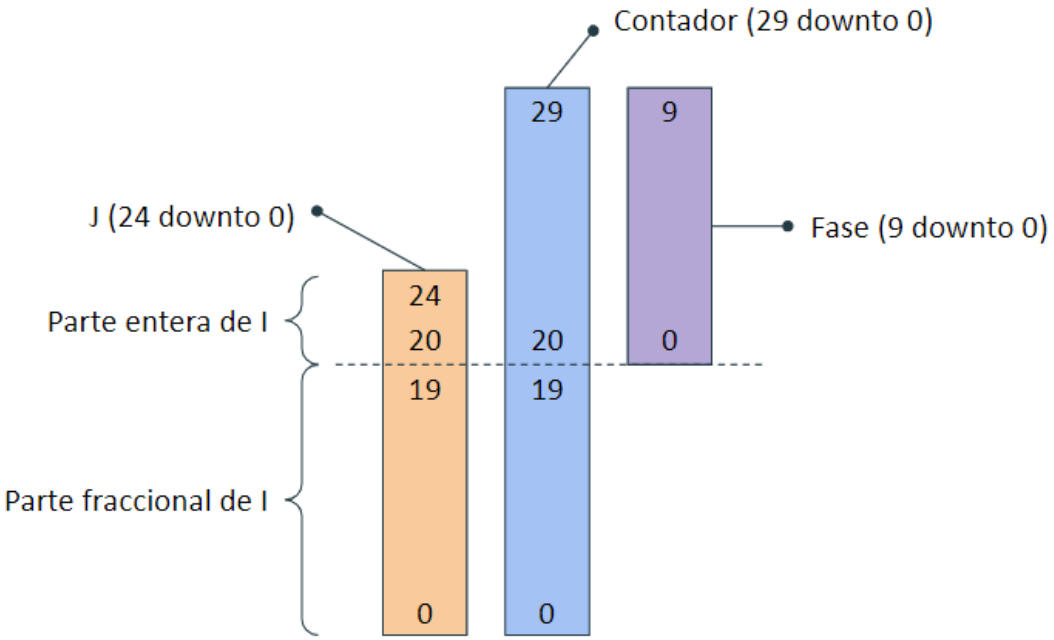

Figura 3. *Correspondencia de los bits del incremento I, el contador y la fase.*

Si deseamos calcular la frecuencia de salida con J, debemos partir de que J es 2^R veces I. Entonces, deberíamos hacer lo siguiente:

$$J = I * 2^R \tag{9}$$

$$F_0 = \frac{F_{Reloj*I}}{2^N} = \frac{F_{Reloj*J}}{2^N * 2^R} = \frac{F_{Reloj}}{2^{N+R}} * J = F_{MIN} * J \tag{10}$$

La ecuación (10) muestra una forma simple de calcular F_0 en función de J y viceversa. Y es mucho más simple trabajar con números enteros para su pasaje entre decimal y binario. De todos modos, tener presente cuándo I es mayor a 1 es importante para saber cuándo la señal de salida se empieza a degradar producto de la disminución de muestras a la salida.

2.2. Conversor Fase-Amplitud

Este bloque tiene como entrada el reset y la fase; y como salida la muestra correspondiente. Además está definido como parámetro la cantidad de bits de la cuantización de la amplitud de la muestra. Esto fue definido como:

```
entity cfa_seno is
    generic (
        Q_BITS_DATO     : positive := 12;       -- Número de bits de cada dato-valor de salida (de 1 a 12)
        N_BITS_MUESTRAS : positive := 10        -- Bits necesario para las N muestras de señal por período
    );
    port (
        reset_i  : in  std_logic;
        fase_i   : in  std_logic_vector(N_BITS_MUESTRAS - 1 downto 0);
        seno_o   : out std_logic_vector(Q_BITS_DATO - 1 downto 0)
    );
end entity cfa_seno;
```

Si `Q_BITS_DATO` es menor a 12, la salida tomará los bits más significativos. La cantidad de bits de la cuantización en el tiempo también está definida como parámetro, pero está implementada sólo para 10 bits.

El núcleo de esta entidad es la definición de su tabla de búsqueda (TdB o LUT). En la misma pueden apreciarse un extracto de las 1024 muestras de 12 bits cada una:

```
    type tdb_type is array (0 to N_MUESTRAS - 1) of std_logic_vector(Q_BITS_TDB - 1 downto 0);
    constant tdb_seno    : tdb_type := (
     "011111111111", "100000001100", "100000011000", "100000100101", "100000110001", "100000111110", "100001001010", "100001010111",
     "100001100011", "100001110000", "100001111101", "100010001001", "100010010110", "100010100010", "100010101111", "100010111011",
     "100011001000", "100011010100", "100011100001", "100011101101", "100011111010", "100100000110", "100100010011", "100100011111",
     "100100101011", "100100111000", "100101000100", "100101010001", "100101011101", "100101101001", "100101110110", "100110000010",
     "100110001110", "100110011011", "100110100111", "100110110011", "100111000000", "100111001100", "100111011000", "100111100100",
     "100111110001", "100111111101", "101000001001", "101000010101", "101000100001", "101000101101", "101000111001", "101001000101",
     "101001010001", "101001011101", "101001101001", "101001110101", "101010000001", "101010001101", "101010011001", "101010100101",
     "101010110001", "101010111101", "101011001000", "101011010100", "101011100000", "101011101100", "101011110111", "101100000011",
```

(…)

```
     "010110101101", "010110111001", "010111000101", "010111010001", "010111011101", "010111101001", "010111110101", "011000000001",
     "011000001101", "011000011010", "011000100110", "011000110010", "011000111110", "011001001011", "011001010111", "011001100011",
     "011001110000", "011001111100", "011010001000", "011010010101", "011010100001", "011010101101", "011010111010", "011011000110",
     "011011010011", "011011011111", "011011101011", "011011111000", "011100000100", "011100010001", "011100011101", "011100101010",
     "011100110110", "011101000011", "011101001111", "011101011100", "011101101000", "011101110101", "011110000001", "011110001110",
     "011110011011", "011110100111", "011110110100", "011111000000", "011111001101", "011111011001", "011111100110", "011111110010"

     );
```

Para un valor de fase_i dado (que debe estar entre 0 y 1023), la tabla de búsqueda devuelve una muestra. Esta entidad fue implementada de forma asincrónica.

Este código y todo el proyecto se encuentra almacenado en el repositorio:

https://github.com/gcapora/CESE_CLP_TP

2.3. Acumulador de Fase

La declaración de la entidad es la siguiente:

```
entity acumulador is
    generic (
        N_BITS_MUESTRAS     : positive := 10;  -- Bits necesario para las N muestras
        J_BITS_INCREMENTO   : positive := 25;  -- Bits del incremento J
        C_BITS_CONTADOR     : positive := 30   -- Bits del contador del AF
    );
    port (
        reloj_i         : in  std_logic;
        reset_i         : in  std_logic;
        habilitar_i     : in  std_logic;
        incremento_i    : in  std_logic_vector(J_BITS_INCREMENTO - 1 downto 0);
        fase_o          : out std_logic_vector(N_BITS_MUESTRAS - 1 downto 0)
    );
end entity acumulador;
```

En el Acumulador de Fase, podemos definir cualquier valor de bits para las muestras de la señal (que en nuestro caso debe ser 10), el incremento J de la resolución en frecuencia y el contador del AF.

El comportamiento del AF es el siguiente:

```
architecture acumulador_arq of acumulador is

    signal contador : unsigned(C_BITS_CONTADOR-1 downto 0);
    constant ceros  : unsigned(C_BITS_CONTADOR-J_BITS_INCREMENTO-1 downto 0) := (others => '0');

begin

    process (reloj_i, reset_i, incremento_i)
    begin
        if reset_i = '1' then               -- El reset es asincrónico
            contador <= (others => '0');
```

```
        elsif rising_edge(reloj_i) then     -- El reloj es sincrónico
            if habilitar_i = '1' then       -- Habilitación para contar
                contador <= contador + (ceros & unsigned(incremento_i));
                -- Si contador desborda, vuelve a contar desde el inicio.
            end if;

        end if;
    end process;

    -- Tomo los bits N_BITS_MUESTRAS más significativos del contador y lo asigno a fase:
    fase_o <= std_logic_vector(contador(C_BITS_CONTADOR-1 downto C_BITS_CONTADOR-
N_BITS_MUESTRAS));

end architecture acumulador_arq;
```

El contador se sincroniza con el reloj de entrada. Ídem con la entrada habilitante. El reset funciona de manera asincrónica. Dejamos que el contador desborde para retomar la fase inicial. Esto nos impone que el CFA tenga una cantidad de muestras que sea una potencia de 2. Por último, se asignan los `N_BITS_MUESTRAS` más significativos del contador a la fase de salida `fase_o`.

2.4. Integración de componentes y simulación

Los bloques CFA y AF se integraron en la entidad seno, definida dentro de `seno.vhd`. Tiene definidos todos los parámetros de las entidades `acumulador` y `cfa_seno`. Tiene como entrada a las entradas de `acumulador`. Y como salida a la salida de `cfa_seno`. Además define una señal para conectar la fase de salida de `acumulador` con la fase de entrada de `cfa_seno`. Todo esto puede verse en la arquitectura de la entidad:

```
architecture seno_arq of seno is

    signal fase : std_logic_vector(BITS_MUESTRAS - 1 downto 0);

begin

    AF: entity work.acumulador
        generic map(
            N_BITS_MUESTRAS   => BITS_MUESTRAS,
            J_BITS_INCREMENTO => BITS_INCREMENTO,
            C_BITS_CONTADOR   => BITS_CONTADOR
        )
        port map(
            reloj_i     =>  s_reloj_i,
            reset_i     =>  s_reset_i,
            habilitar_i =>  s_habilitar_i,
            incremento_i=>  s_incremento_i,
            fase_o      =>  fase
        );

    CFA: entity work.cfa_seno
        generic map(
            Q_BITS_DATO => BITS_DATO
        )
        port map(
            reset_i =>  s_reset_i,
            fase_i  =>  fase,
            seno_o  =>  s_seno_o
        );

end architecture seno_arq;
```

La simulación fue hecha inicialmente con GHDL y GTKwave. En la figura se muestra una simulación para una señal de 10 kHz. Las ventajas de GHDL y GTKwave son su simplicidad y

rapidez de utilización. Su utilización no exige especificar un FPGA sino que prueba la interconexión de componentes funcionales.

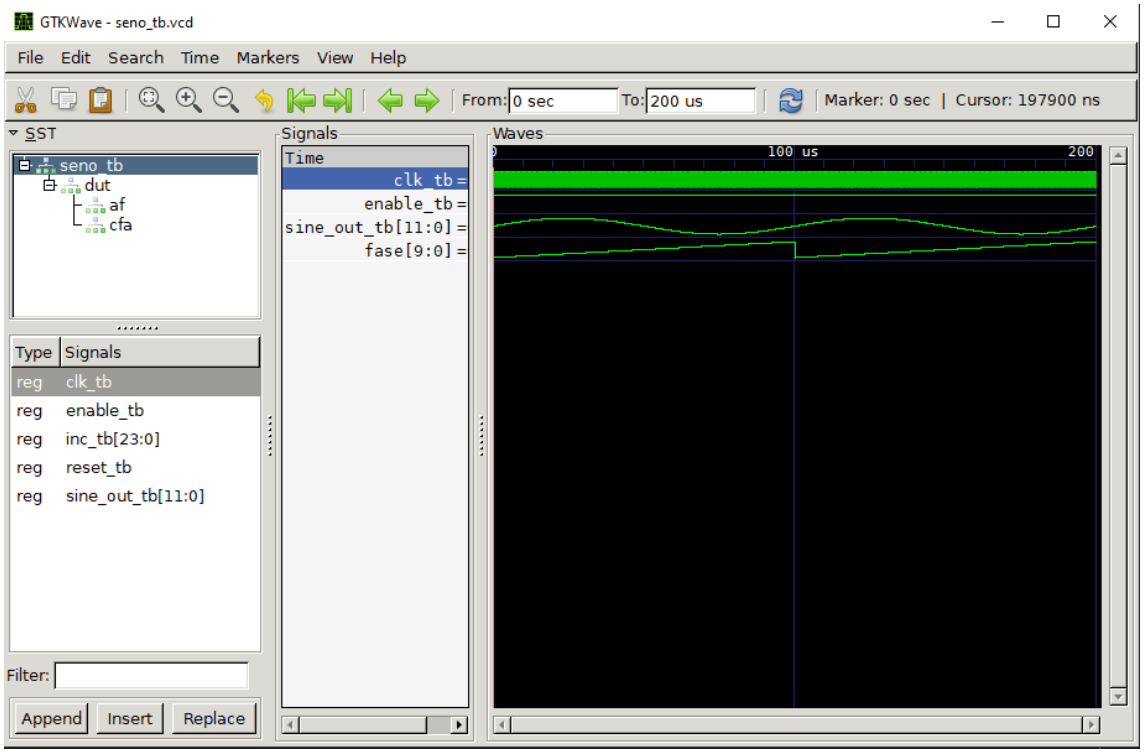

Figura 4. *Simulación en GTKwave para una señal de 10 kHz.*

En la simulación se configuró la frecuencia calculada en la ecuación (8). Como se aprecia en la figura, la frecuencia tiene un período de 100 us, o 10 kHz, tal como se había buscado.

2.5. Síntesis e implementación para prueba en Arty Z7-10

Para la prueba en la placa abrimos un proyecto en Vivado y cargamos los archivos fuente .vhd, el archivo de simulación que prueba las fuentes, la plantilla de restricciones y seleccionamos la placa Arty Z7-10. Empezamos por el RTL Analysis. En las figuras 5, 6 y 7 se muestran los esquemáticos de conexiones:

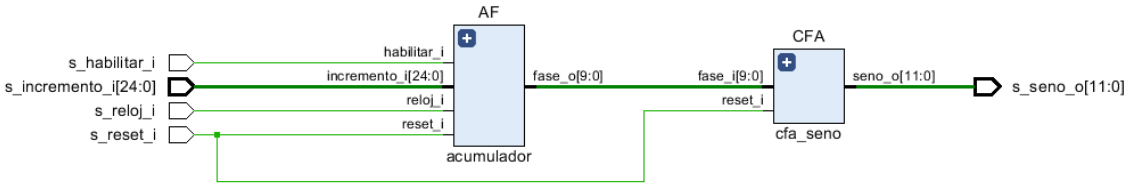

Figura 5. *Diagrama esquemático del NCO senoidal.*

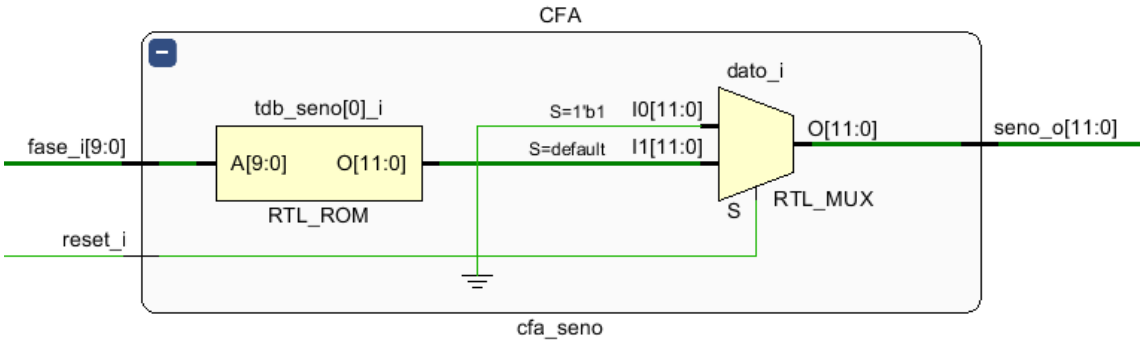

Figura 6. *Detalle del esquemático del CFA.*

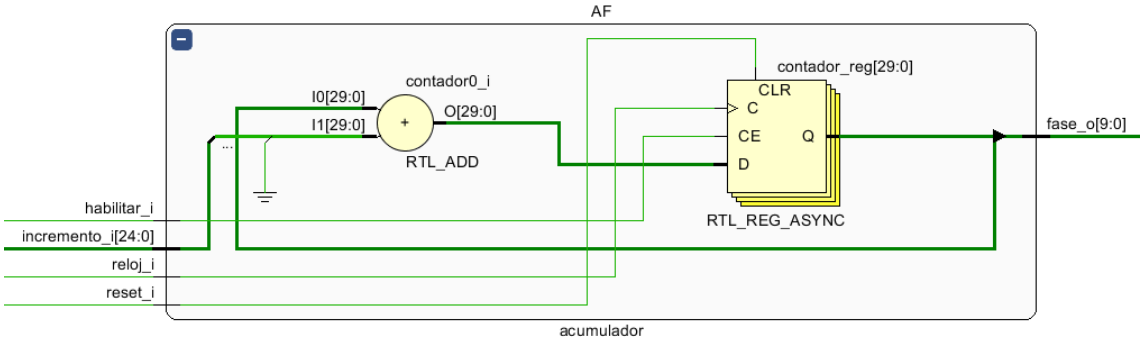

Figura 8. *Detalle del esquemático del AF.*

Luego continuamos con la simulación de comportamiento en Vivado, que resulta ser similar a la realizada previamente. Lo probamos para 1 MHz:

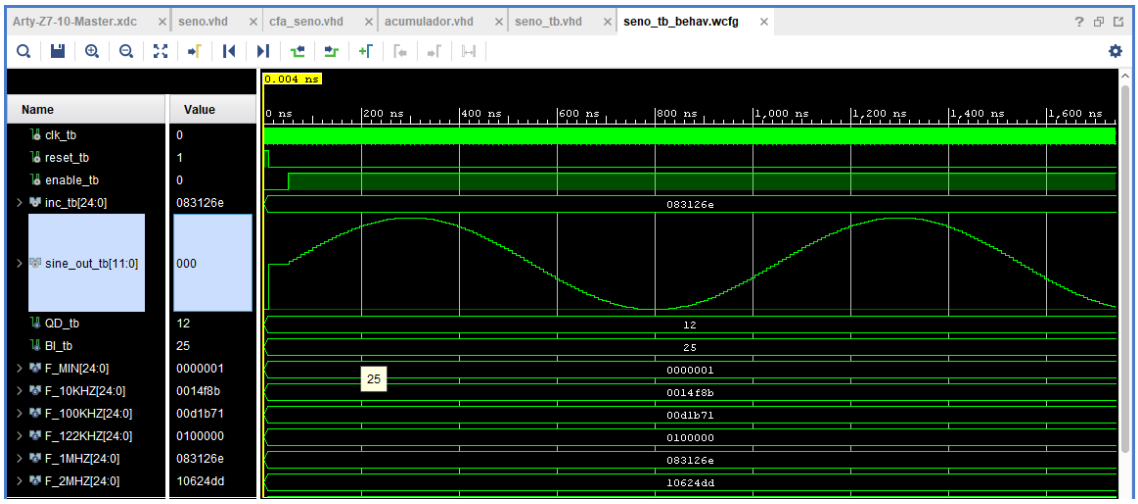

Figura 9. *Simulación en Vivado para F_0 = 1 MHz.*

Entonces comenzamos con la Synthesis y luego la Implementation. La figura 10 muestra el esquemático de la Synthesis. Los componentes lógicos de las figuras anteriores fueron reemplazados por los que efectivamente tiene la placa a utilizar.

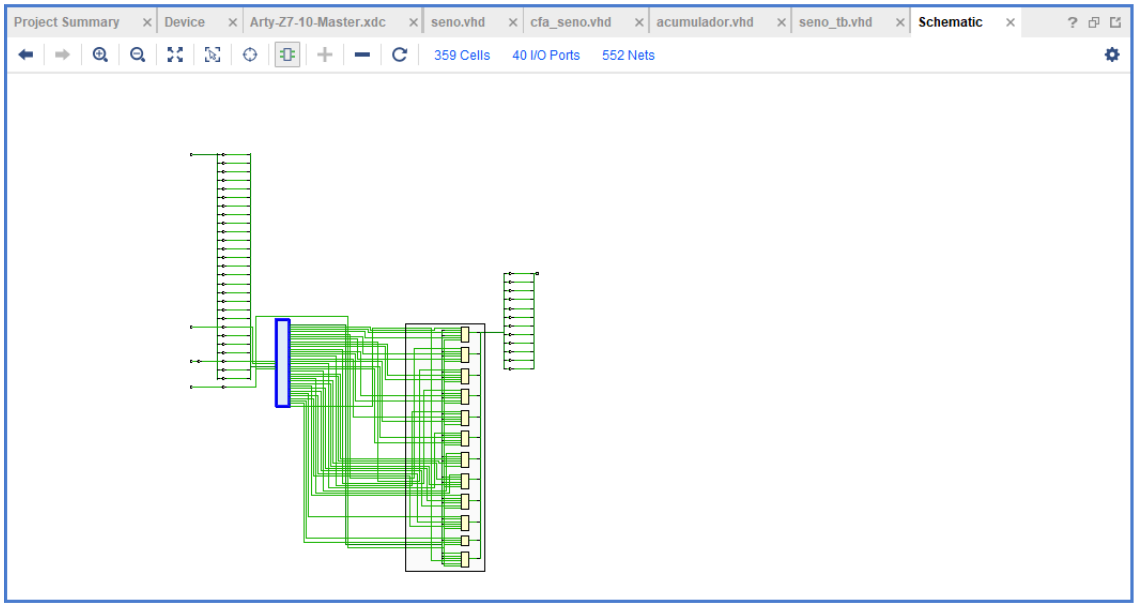

Figura 10. *Esquema del sistema sintetizado, con el CFA ampliado.*

2.6. Utilización de IP cores para prueba remota

En este punto, necesitamos utilizar alguna de las entidades de procesamiento de propiedad intelectual (IP core) para probar el desarrollo en una placa física concreta. En primera instancia utilizamos el IP core VIO. Los archivos utilizados para programar la placa son:

```
Vivado_VIO\NCO senoidal VIO.runs\impl_1\seno_VIO.bit
Vivado_VIO\NCO senoidal VIO.runs\impl_1\seno_VIO.ltx
```

Tras abrir el servidor y programar la placa, pudimos visualizar el funcionamiento en la siguiente ventana:

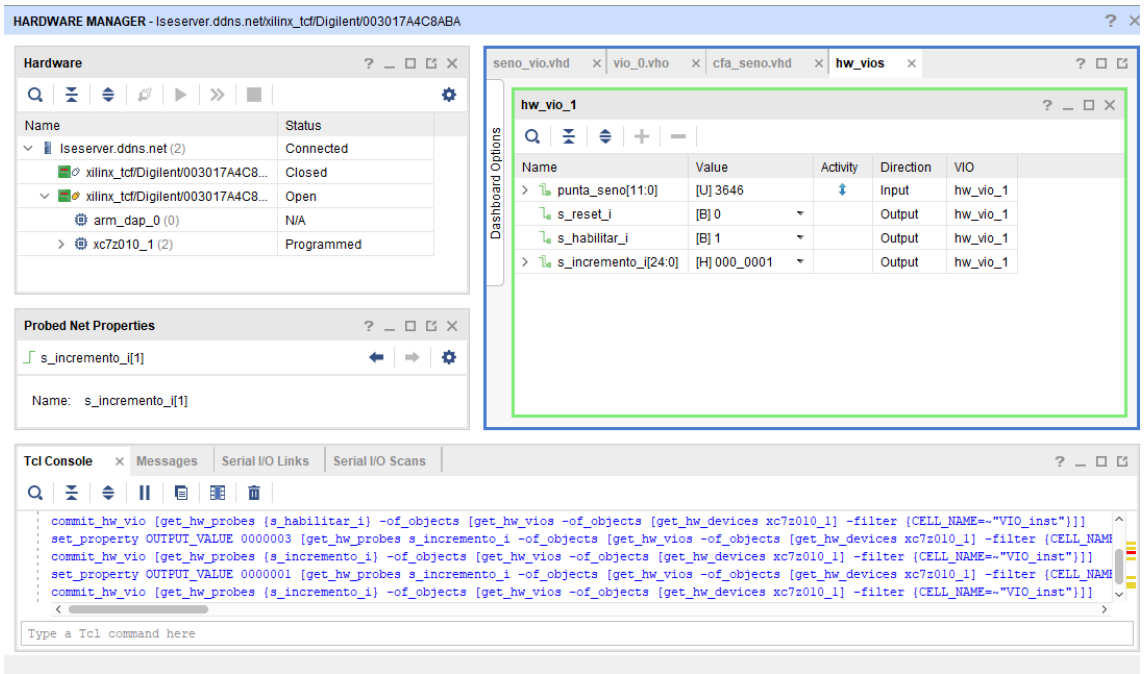

Figura 11. *Probando oscilador con VIO a una F_0 = 0,11 Hz.*

Dado que la interfaz VIO actualiza la información para tiempos bastantes largos, programamos al NCO en su frecuencia más baja de 0,11 Hz. Esto nos permitió verificar que la implementación funciona, aunque no nos muestra la forma de la señal senoidal.

Lo que siguió fue incorporar un IP core ILA. El mismo nos permite mostrar señales de la placa como si fuera un analizador de datos. Además, permite mostrar los vectores de bits como números decimales o en forma analógica. Con el core VIO configuramos la frecuencia y con el core ILA analizamos la señal. La figura 12 nos da una de las señales.

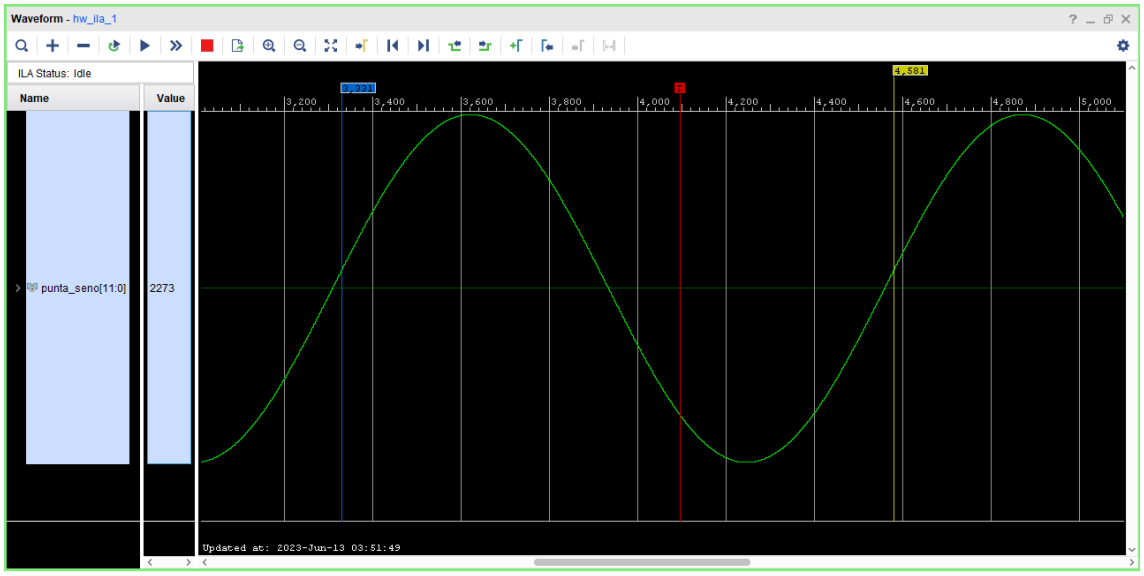

Figura 12. *Probando oscilador con ILA a una F_0 = 100 kHz.*

En la señal captamos el momento en que vuelve a pasar por el mismo punto. La diferencia de muestras nos dio 1250 muestras. Esta cantidad de muestras por el período del reloj resultó ser:

$$T_0 = 1250 * T_{Reloj} = \frac{1250}{F_0} = 10\ \mu s \tag{11}$$

$$F_0 = \frac{1}{T_0} = \frac{F_{Reloj}}{1250} = 100\ \text{kHz} \tag{12}$$

En la figura 13, configuramos una frecuencia de 2 MHz. En la imagen, ya puede verse el efecto de la discretización en el tiempo. Esto no es deseable pero es lo mejor posible para una frecuencia de reloj de 125 MHz.

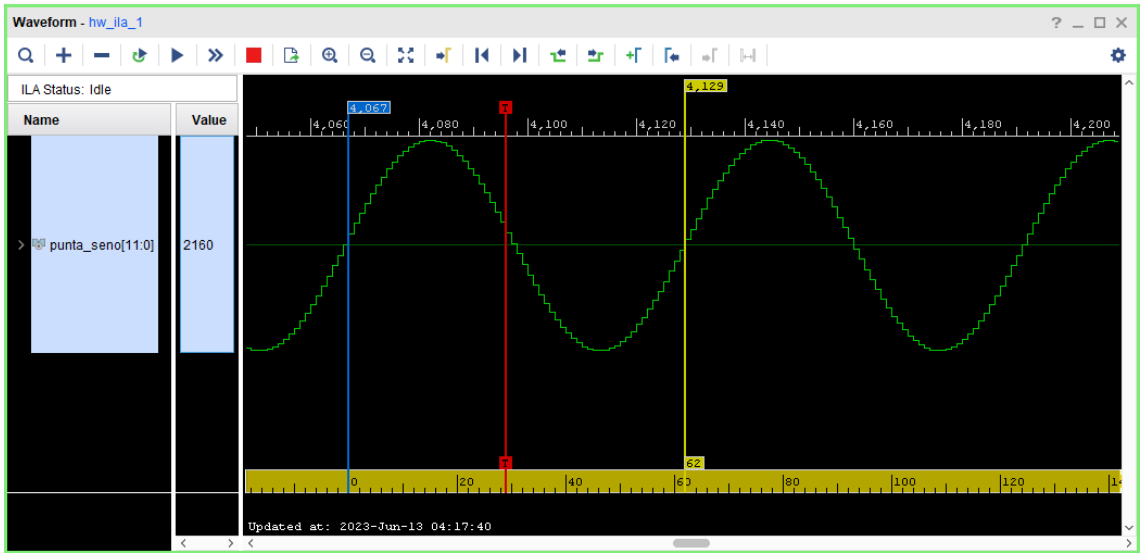

Figura 13. *Probando oscilador con ILA a una F_0 = 2 MHz.*

3. Conclusiones

A lo largo del presente trabajo hemos presentado el diseño, la implementación y la prueba física de nuestro NCO senoidal. Hemos logrado configurar una frecuencia variable y establecer una salida de 12 bits utilizando una tabla de búsqueda. La prueba en prototipo se ha hecho de forma remota y ha resultado exitosa.

Uno de los aspectos tan poderosos de la FPGA es que permite configurar diversas entidades y procesos que, de hecho, funcionan como unidades de procesamiento (cores) concurrentes. Es decir: hay un procesamiento en paralelo y no es serie, como es lo que ocurre con un microprocesador de núcleo único.

Una mejora que podría tener este desarrollo es lograr implementar toda la señal a partir de un cuarto de onda, aprovechando las simetrías de la señal seno. Hemos utilizado una cantidad de muestras direccionables con 10 bits. Si implementáramos la señal completa a partir del cuarto de onda, utilizaríamos casi la misma cantidad de TdB's (o LUT's) pero con un muestreo de 12 bits, en sintonía con la cuantización en amplitud de 12 bits.

4. Documentos y referencias

[1] Álvarez, Nicolás. “NCO (Numerically Controlled Oscillator)” (presentación), Sistemas Digitales 66.17, Facultad de Ingeniería, UBA.

[2] Álvarez, Nicolás (2018). “Circuitos lógicos programables: Ciclo de desarrollo con Vivado (Práctica 1)”, curso de Circuitos Lógicos Programables, CESE, Facultad de Ingeniería, UBA.

[3] Repositorio del proyecto: https://github.com/gcapora/CESE_CLP_TP.

[4] Wikipedia, “Oscilador de cristal”. Consultado en: https://es.wikipedia.org/wiki/Oscilador_de_cristal (junio de 2023).

5. Definiciones, acrónimos y abreviaturas

1. ADC Conversor Analógico-Digital
2. CESE Carrera de Especialización en Sistemas Embebidos
3. DAC Conversor Digital-Analógico
4. FIUBA Facultad de Ingeniería de la Universidad de Buenos Aires
5. FPGA Matriz de compuertas lógicas programables (*Field Programable Gate Array*)
6. LUT *Look-up table*, equivalente a TdB
7. N/A No aplica
8. ppm Parte por millón
9. TdB Tabla de búsqueda, equivalente a LUT