Trabajo práctico final
Circuitos Lógicos Programables

Especialización en Sistemas Embebidos

NCO

(Oscilador controlado numéricamente)

Alumno: Ing. Leandro Arrieta

Profesor: Ing. Nicolás Álvarez

Fecha: 18/04/2022

1.Introducción **3**

2.Funcionamiento **3**

2.1. Registro de control de frecuencia 4

2.2. Memoria senoidal 5

2.3. Bloque atenuador 6

2.4. Bloque principal 6

3.Simulación **6**

3.1 Simulación con GTKWave 6

3.2. Simulación con Vivado **7**

4. Síntesis e implementación en Vivado **7**

4.1.Esquemático 8

4.2.Uso de recursos de la FPGA 8

5. Prueba con ILA y VIO **9**

1.Introducción

Se diseñó un oscilador de onda senoidal controlado numéricamente al cual se le puede ajustar la frecuencia y la amplitud de salida para funcionar en la placa de desarrollo de Xilinx Arty Z7-10.

Para el desarrollo y prueba del programa se utilizó en primer instancia un editor de texto y el software de simulación GTKWave. Luego se paso a Vivado donde se volvió a simular, se realizo la sintesis, implementación y se generó el bitstream. Para probar el sistema se hizo uso de las herramientas que provee Vivado y la FPGA, se usó un VIO (Virtual Input Output) para seleccionar la frecuencia y atenuación de la señal de salida y de un ILA (Integrated Logic Analizer) para poder visualizar la salida y probar el bitstream directamente en la FPGA.

Durante el diseño se tuvo la premisa de no dejar valores hardcodeados y que con solo cambiar una variable se pueda fácilmente cambiar cualquiera de los parámetros de diseño como ser cantidad de muestras, bits y nivel de atenuaciónde la señal de salida.

2.Funcionamiento

El trabajo se divide en tres bloques principales.

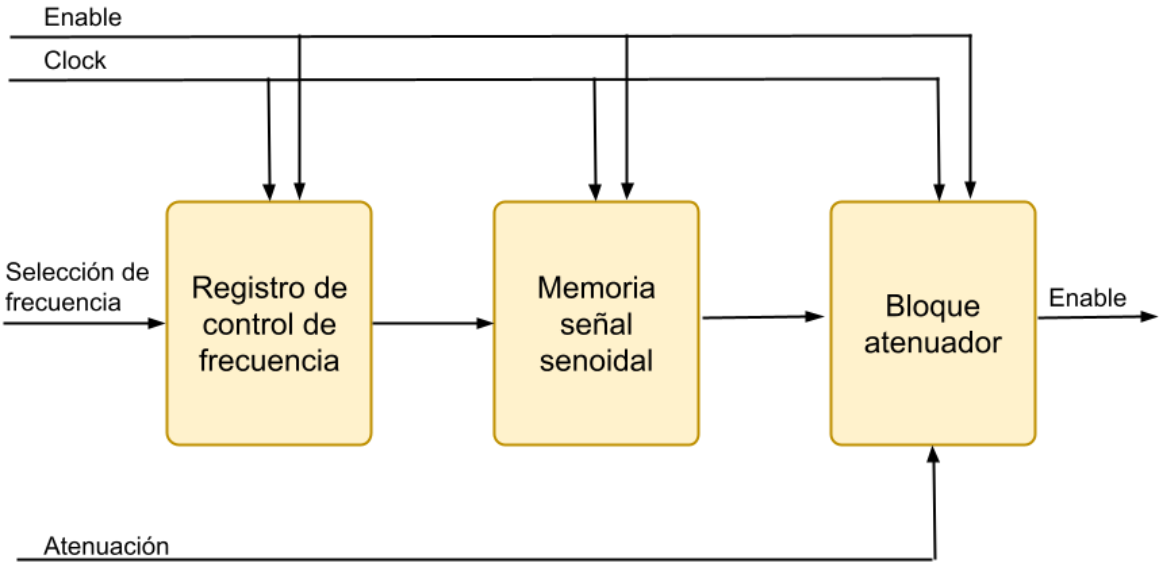


Fig 1. Diagrama en bloques

Todos los bloques reciben la señal de Clock y el enable de habilitación. Cada uno de los bloques se desarrolló en un archivo aparte. Luego se juntaron todos estos archivos como componentes en el programa principal nombrado NCO.vhd.

2.1. Registro de control de frecuencia

Está implementado en el archivo principal RCF.vhd.

Es el encargado de controlar la frecuencia de la señal de salida.

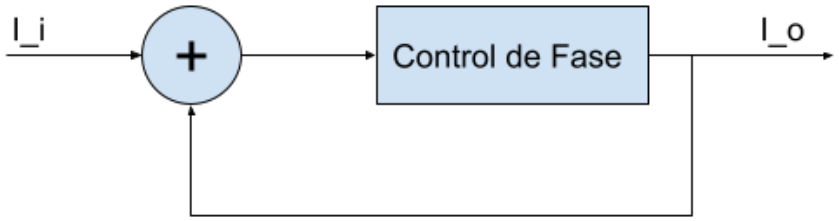


Fig 2. Estructura interna RCF.vhd

Recibe a la entrada la señal **I_i,** el control de fase inicia en cero y con cada pulso de clock se incrementará según el valor de **I_i**. Cuanto más grande sea el valor de **I_i** más rápido crecerá el control de fase por lo que tendremos una mayor frecuencia de la señal de salida. La salida de este bloque, **I_o**, se conecta a la memoria que contiene la señal senoidal.

De lo anterior se desprende que:

$$Fo = \frac{Fclk \times I\_i}{M}$$

Siendo M la cantidad de muestras en la memoria de la señal de salida

A continuación vemos el código del bloque *process* del control de frecuencia.

```
process (clk_i)
    variable addr_aux : integer range 0 to (2**(N_RCF+1)):=0;
begin
    if rising_edge(clk_i) then
        if ena_i = '1' then
            addr_aux := addr_aux + to_integer(unsigned(I_i));
            if addr_aux >= (2**N_RCF) then
                addr_aux := addr_aux -(2**N_RCF);
            end if;
            I_o <= std_logic_vector(to_unsigned(addr_aux,N_RCF));
        end if;
    end if;
end process;
```

Fig 3. Bloque process de RCF.vhd

Para hacer las operaciones algebraicas se usó la librería **ieee.numeric_std** convirtiendo el valor de **I_i** a un entero, operarlo junto con una variable auxiliar y una vez que se tenía el valor de la señal de salida se volvió a convertir a un **std_logic_vector**.

Para mantener la continuidad de la señal de salida, cuando el valor de la señal auxiliar **addr_aux** desborda su tamaño máximo, en vez de volverlo a cero lo que se hace es restarle su valor máximo. Se hace esto para evitar que ante valores de **I_i** mayores a uno se genere algún glich en la señal de salida. Se dejó todo referenciado a un valor genérico **N_RCF** para fácilmente poder cambiar el tamaño a direccionar de la memoria.

2.2. Memoria senoidal

La memoria se armó como un componente aparte para facilitar la modularidad, si bien trabajamos con una sola señal de salida, con este esquema sería relativamente sencillo agregar otra memoria con otra forma de onda.

Este bloque recibe en **addr_i** el valor de **address** del bloque de registro de control de frecuencia, ante un flanco ascendente del clock selecciona el valor de salida que le corresponde para ese address.

La memoria se preparó con 256 muestras de 10 bits cada una, pero se puede ajustar a la cantidad de muestras y bits que uno quiera.

```
process (clk_i)
    -- Le doy un bit mas para evitar error por desborde
    variable address: integer range 0 to (2**(N_mem+1));
begin
    if rising_edge(clk_i) then
        if ena_i = '1' then
            address := to_integer(unsigned(addr_i));

       if address >= (2**N_mem) then address := address -(2**N_mem);
            end if;

            if    address= 0    then mem_out <= "0111111111";
            elsif address= 1    then mem_out <= "1000001100";
            elsif address= 2    then mem_out <= "1000011000";
                                .
                                .
            elsif address= 255  then mem_out <= "0111110010";
     elsif address>= (2**N_mem) then address := address -(2**N_mem);
            end if;
        end if;
    end if;
end process;
```

Fig 4. Bloque process de mem_seno.vhd

2.3. Bloque atenuador

Este bloque está implementado en el archivo att.vhd, recibe los datos de la memoria, y de acuerdo al valor de atenuación seleccionado divide el dato de la memoria por el valor de atenuación. Al hacer uso de la librería **ieee.numeric_std** la operación de dividir resulta muy sencilla. Al igual que los otros bloques, actúa ante un flanco ascendente del clock y si se encuentra la señal de enable en alto.

2.4. Bloque principal

Finalmente los tres bloques anteriores se utilizan como componentes en la arquitectura del programa principal que se encuentra en el archivo NCO.vhd, este archivo simplemente instancia los 3 bloques, le da valores a los generics y hace uso de algunas señales auxiliares para conectar los distintos bloques entre sí.

3.Simulación

3.1 Simulación con GTKWave

Para simular el programa se hizo uso del software GTKWave. El banco de pruebas se armó en el archivo NCO_tb.vhd. A continuación podemos ver una captura de pantalla donde tenemos en orden las siguientes señales, clock, enable, atenuación, control de frecuencia, y salida. Para visualizar de forma más agradable y entendible la salida se seleccionó visualizar esta señal en forma analógica.

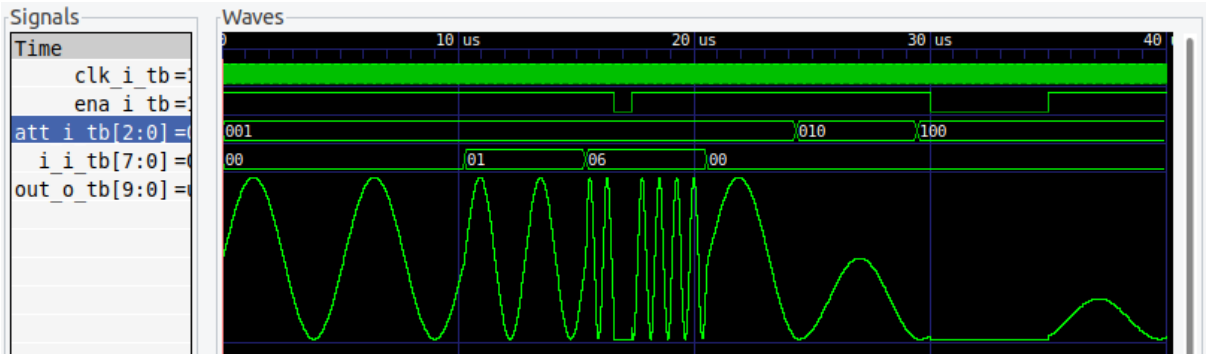


Fig 5. Simulación con GTKWave

Los parámetros de las señales de simulación los podemos ver en las siguientes líneas del archivo NCO_tb.vhd

```
clk_i_tb <= not clk_i_tb after 10 ns;
ena_i_tb <= '0' after 16600 ns, '1' after 17330 ns,
            '0' after 30000 ns, '1' after 35000 ns;
att_i_tb <= std_logic_vector(to_unsigned(2,j_tb)) after 24320 ns,
            std_logic_vector(to_unsigned(4,j_tb)) after 29440 ns;
I_i_tb   <= std_logic_vector(to_unsigned(1,N_tb)) after 10240 ns,
            std_logic_vector(to_unsigned(6,N_tb)) after 15360 ns,
```

Fig 5. Configuración señales en NCO_tb.vhd

En la simulación podemos ver cómo al cambiar la señal **i_i_tb** cambia la frecuencia de la señal, cómo con la variación de **att_i_tb** cambia la amplitud y con **ena_i_tb** se habilita o deshabilita el circuito.

En la siguiente imágen podemos ver cómo los cambios de la señal de salida se dan en flancos ascendentes de clock y que la señal de enable cae en medio del pulso de clock y recién toma efecto ante el flanco ascendente del clock

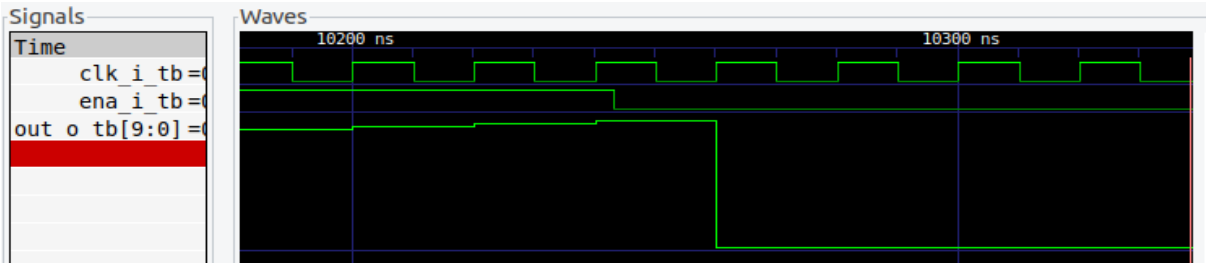


Fig 6. Cambios de salida sincronizados por el clock

3.2. Simulación con Vivado

Una vez que se tenía el programa funcionando en GTKWave, se lo llevó a vivado para realizar la síntesis, implementación y generar el Bitstream, se repitió la simulación en Vivado y los resultados fueron los mismo a los obtenidos con el GTKWave.

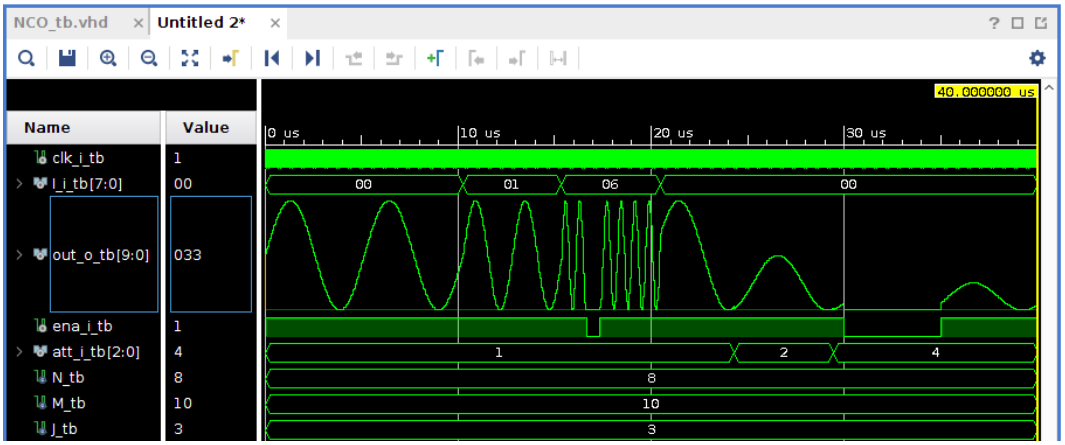
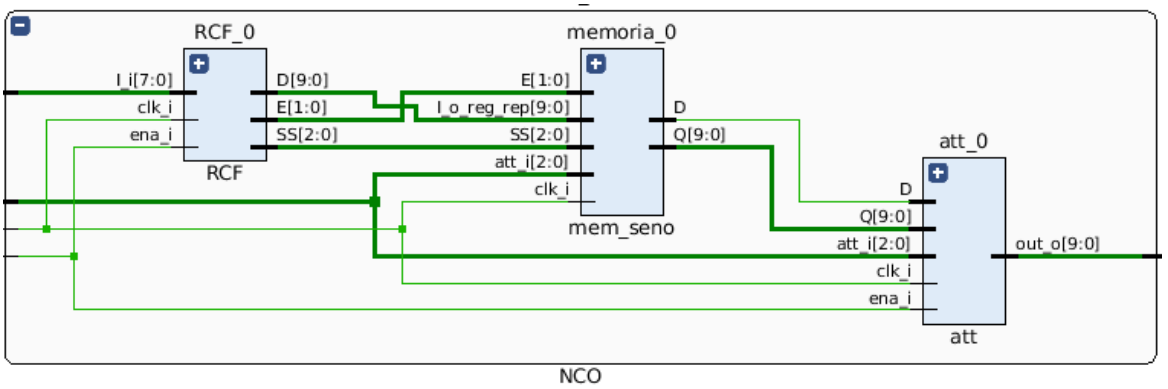


Fig 7. Simulación con Vivado

4. Síntesis e implementación en Vivado

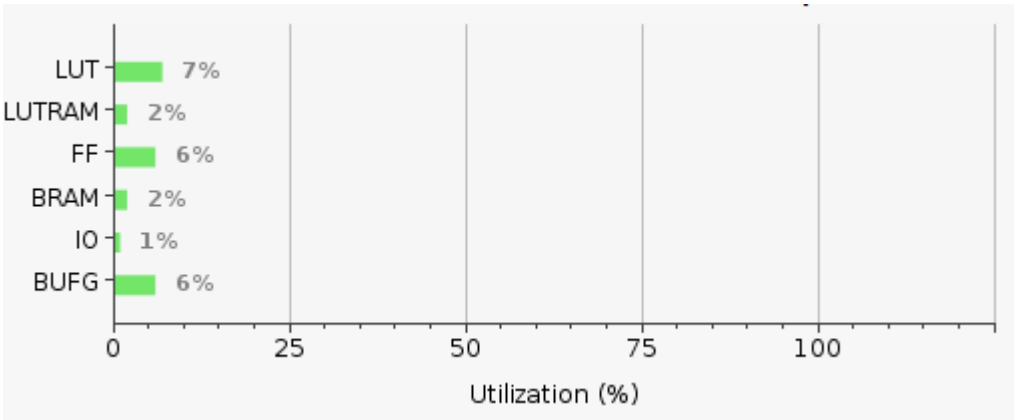
Para probar el programa en la FPGA se creó un nuevo archivo NCO_IP.vhd el cual contiene todos los archivos vistos hasta el momento y se instanció un VIO y un ILA.

4.1.Esquemático



4.2.Uso de recursos de la FPGA

Resource	Utilization	Available	Utilization %
LUT	1297	17600	7.37
LUTRAM	94	6000	1.57
FF	2148	35200	6.10
BRAM	1	60	1.67
IO	1	100	1.00
BUFG	2	32	6.25



5. Prueba con ILA y VIO