.UBAfiuba
FACULTAD DE INGENIERÍA

Trabajo práctico Final

Microarquitecturas y Softcores

Docente: Ing. Nicolás Álvarez
Autor: Abraham Rodriguez
Email: abraham.rodz17@gmail.com

Diciembre, 2024

Índice

1. Funcionalidad e implementación

En esta sección se proporciona una breve explicación del IP core de un NCO implementado e implmentado mediante, así como diagramas y bloques de código relevantes.

1.1. Repositorio

La NCO implementada se encuentra en el repositorio:
https://github.com/AbeRodz/Numeric-Controlled-Oscillator
El trabajo realizado implementada se encuentra en el repositorio:
https://github.com/AbeRodz/Zynq-NCO

1.2. Numeric Controlled Oscillator (NCO)

Un NCO consiste de un acumulador de fase y convertidor de fase de amplitud (Generador de ondas), generalizando en un diagrama corresponde a la imagen siguiente:

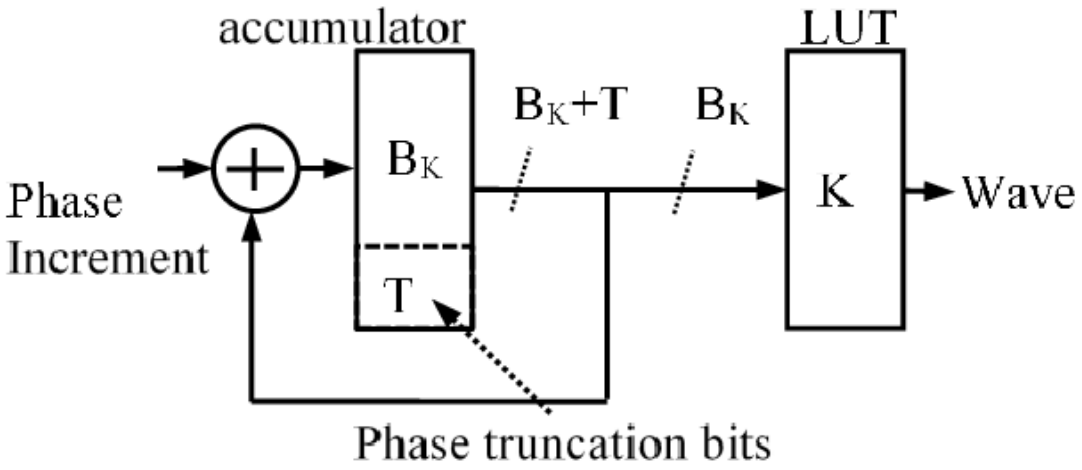

Figura 1: Diagrama interno NCO.

El generador de ondas es comúnmente una LUT con los valores de la función seno, sin embargo existen algoritmos para generar los valores como:

- CORDIC
- Aproximación de Taylor
- Aproximación lineal
- Direct Digital Synthesis (DDS)

La NCO es capaz de generar distintos tipos de onda, en este caso se implementaron las ondas:

- Seno
- Cuadrada
- Triangular
- Sierra

Lo que difiere entre todos los métodos es la cantidad de memoria utilizada, la velocidad y la precisión. En el presente trabajo se implemento un generador basado en LUT, lo cual implica que el consumo de memoria es alto a comparación de otros métodos.

1.2.1. Acumulador

El acumulador es muy simple, consiste en aplicar la operación de suma de la fase o *Frequency Tuning/Control Word* actual con el estado anterior. En codigo simplificado VHDL del acumulador se ve de la siguiente manera:

```
1
2   process (c_i)
3   begin
4     if rising_edge(c_i) then
5       if en_i = '1' then
6         -- Phase accumulator update
7         phase_acc <= phase_acc + unsigned(freq_word_i);
8
9         -- Extract upper bits of the phase accumulator for the SineLUT
10        addr <= std_logic_vector(phase_acc(31 downto 22));
11      end if;
12    end if;
13  end process;
```

Listing 1: Acumulador

La variable phase_acc contiene el valor acumulado en 32 bits, sin embargo la implementación de sineLUT contiene una LUT de 1024 registros (10bits) en otras palabras una muestra discreta de 1024 registros de una función senoidal, por lo tanto se extraen los bits mas significativos para ser mapeados a la LUT.

1.2.2. SineLUT

Para generar los valores de la funcion seno, se realizo un script de Python y se guardo a un archivo .txt para luego ser escritos al modulo de la LUT. El script consiste en:

```
1
2 def generate_uint16_sine_table(num_samples: int = 1024, amplitude: int = 32767,
      offset:int = 32767) -> tuple[np.ndarray]:
3     num_values = num_samples
4     x = np.linspace(0, 2 * np.pi, num_values, endpoint=False)
5     y = np.round(amplitude * np.sin(x) + offset).astype(int)
6     return x,y
```

Listing 2: Acumulador

En este caso la LUT contiene valores de la función seno con los siguientes parámetros:

- 1024 muestras.
- Tipo uint16.

En otras palabras la LUT contiene 10x16bits.
Al graficar la muestra de la función seno obtenida se obtiene:

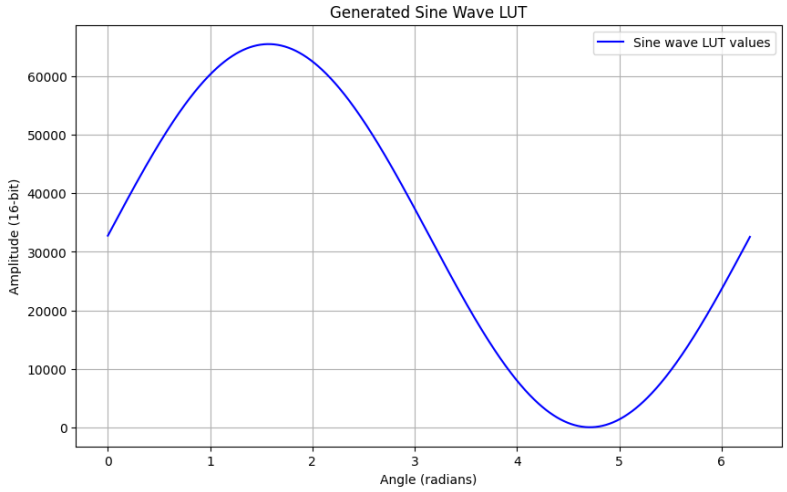

Figura 2: Muestra de la función seno.

Una vez generados los valores, se creo el modulo sineLUT.vhd, el cual guarda en memoria una variable llamada ROM con los valores de la muestra. Para buscar los valores dentro de la tabla, se debe de buscar un registro dentro de la tabla, este registro es calculado del acumulador.

```
1
2  begin
3
4    SIN : process (c_i)
5    begin
6      if rising_edge(c_i) then
7        if (en_i = '1') then
8          wave_o <= std_logic_vector(to_Signed((t_ROM(to_Integer(Unsigned(addr_i))) -
       32768), 16));
9        end if;
10     end if;
11   end process SIN;
12
13 end;
```

Listing 3: SineLUT

1.2.3. NCO

La NCO consiste en una instancia de sineLUT y una signal como acumulador, la funcionalidad basica es dada por el bloque de código mostrado anteriormente 1.2.1.

La NCO implementada en el presente trabajo es capaz de generar varios tipos de onda utilizando postprocesamiento de la señal dada por la sineLUT y el acumulador.

```
1
2  begin
3     if rising_edge(c_i) then
4       if en_i = '1' then
5         -- Phase accumulator update
6         phase_acc <= phase_acc + unsigned(freq_word_i);
7
8         -- Extract upper bits of the phase accumulator for the SineLUT
9         addr <= std_logic_vector(phase_acc(31 downto 22));
10
11        -- Generate square wave (based on MSB of phase accumulator)
12        if phase_acc(31) = '1' then
13          square_wave <= std_logic_vector(to_signed(32767, 16)); -- High output
14        else
```

```
15            square_wave <= std_logic_vector(to_signed(-32767, 16)); -- Low output
16          end if;
17
18          -- Generate triangle wave
19          if phase_acc(31) = '1' then
20            -- Descending part of the triangle wave (negative half)
21            triangle_wave <= std_logic_vector(to_signed(32767 - to_integer(unsigned(
    phase_acc(30 downto 16))), 16));
22          else
23            -- Ascending part of the triangle wave (positive half)
24            triangle_wave <= std_logic_vector(to_signed(to_integer(unsigned(phase_acc
    (30 downto 16))), 16));
25          end if;
26
27          -- Generate sawtooth wave (directly from phase accumulator)
28          sawtooth_wave <= std_logic_vector(phase_acc(31 downto 16));
29
30          -- Select the output waveform based on wave_type_i
31          case wave_type_i is
32            when "00" =>
33              wave_o <= sine_wave; -- Output sine wave
34            when "01" =>
35              wave_o <= square_wave; -- Output square wave
36            when "10" =>
37              wave_o <= triangle_wave; -- Output triangle wave
38            when "11" =>
39              wave_o <= sawtooth_wave; -- Output sawtooth wave
40            when others =>
41              wave_o <= sine_wave; -- Default to sine wave
42          end case;
43        end if;
44      end if;
45    end process;
```

Listing 4: SineLUT

1.2.4. Integración con microprocesador

Una vez creado el NCO, se empaqueto como un IP Core mediante el package manager.

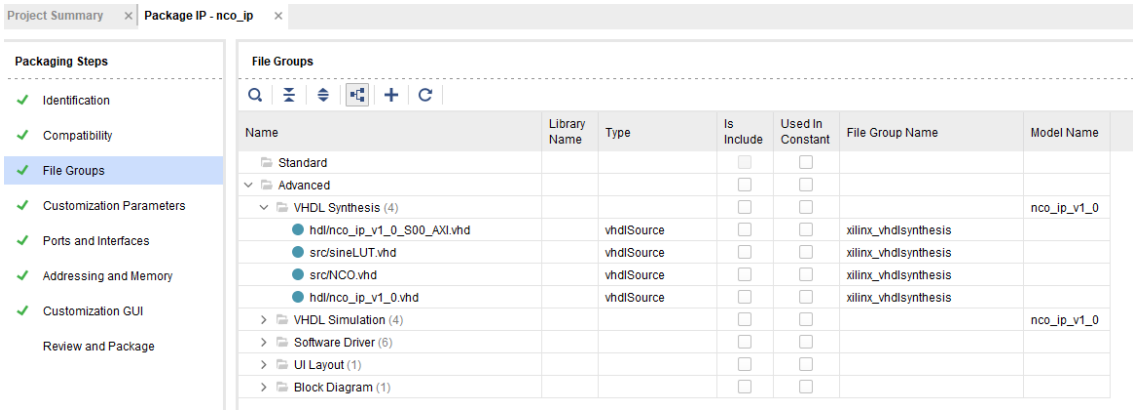

Figura 3: NCO IP Package.

Crear el IP Core de la NCO implica exponer el output de la NCO hacia el exterior para poder ser analizado mediante una ILA. Para eso se agrego al port la variable de salida.

```
1
2  entity nco_ip_v1_0 is
3    generic (
4      -- Users to add parameters here
5
6      -- User parameters ends
7      -- Do not modify the parameters beyond this line
8
9
10     -- Parameters of Axi Slave Bus Interface S00_AXI
11     C_S00_AXI_DATA_WIDTH   : integer := 32;
12     C_S00_AXI_ADDR_WIDTH   : integer := 4
13   );
14   port (
15     -- Users to add ports here
16         wave_output  : out std_logic_vector(C_S00_AXI_DATA_WIDTH-17 downto 0); --
     Output wave from NCO
```

Listing 5: NCO$_IP$

Para luego ser propagado hacia la instancia de la NCO.

```
1
2      inst_NCO: entity work.NCO
3      port map (
4        c_i          => S_AXI_ACLK,
5        en_i         => slv_reg3(0),
6        wave_type_i  => wave_type_i,
7        freq_word_i  => slv_reg0,
8        wave_o       => wave_output
9      );
```

Listing 6: NCOInst

Dando como resultado el siguiente bloque:

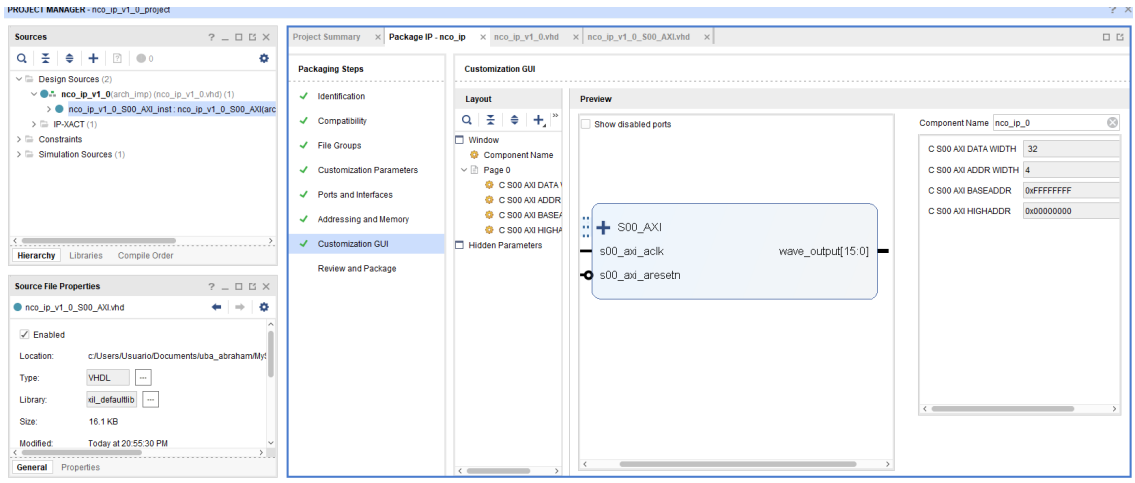

Figura 4: NCO GUI Package.

Una vez creado el paquete de la NCO, se realizo el siguiente código C mediante la SDK:

```
1
2  /*
3   * nco.c
4   *
5   *  Created on: Nov 30, 2024
6   *      Author: Abraham R.
```

```
 */

#include "xparameters.h"
#include "xil_io.h"
#include "nco_ip.h"


/*
 * REG 0 ->  frequency word
 * REG 1 -> wave type
 * REG 2 -> wave output
 * REG 3 -> enable
 *
 */

#define NCO_BASE_ADDR XPAR_NCO_IP_0_S00_AXI_BASEADDR
#define REG_FREQUENCY_OFFSET 0x00
#define REG_WAVE_TYPE_OFFSET 0x04
#define REG_WAVE_OUTPUT_OFFSET 0x08
#define REG_ENABLE_OFFSET 0x0C


// NCO configuration and state
typedef struct {
    int32_t base_addr;        // Base address of the NCO
    int32_t frequency;        // Current frequency tuning word
    int32_t wave_type;        // Current wave type
    int32_t enable;           // Enable status (0 or 1)
    int32_t freq_step;        // Step size for frequency adjustment
    int32_t min_freq;         // Minimum frequency tuning word
    int32_t max_freq;         // Maximum frequency tuning word
} NCO_Config;

// Function to initialize NCO settings
void NCO_Init(NCO_Config *nco, int32_t base_addr, int32_t init_freq, int32_t
    freq_step, int32_t min_freq, int32_t max_freq) {
    nco->base_addr = base_addr;
    nco->frequency = init_freq;
    nco->wave_type = 0x01; // Default to sine wave
    nco->enable = 0x01;   // Enable by default
    nco->freq_step = freq_step;
    nco->min_freq = min_freq;
    nco->max_freq = max_freq;
}

// Function to write frequency tuning word
void NCO_SetFrequency(NCO_Config *nco, int32_t frequency_tuning_word) {
    nco->frequency = frequency_tuning_word;
    NCO_IP_mWriteReg(nco->base_addr, REG_FREQUENCY_OFFSET, nco->frequency);
    xil_printf("Frequency tuning word set to: 0x%08X\r\n", nco->frequency);
}

// Function to set wave type
void NCO_SetWaveType(NCO_Config *nco, int32_t wave_type) {
    nco->wave_type = wave_type;
    NCO_IP_mWriteReg(nco->base_addr, REG_WAVE_TYPE_OFFSET, nco->wave_type);
    xil_printf("Wave type set to: %u\r\n", nco->wave_type);
}

// Function to enable/disable NCO
void NCO_SetEnable(NCO_Config *nco, int32_t enable) {
    nco->enable = enable;
    NCO_IP_mWriteReg(nco->base_addr, REG_ENABLE_OFFSET, nco->enable);
    xil_printf("Enable signal set to: %u\r\n", nco->enable);
}

// Function to dynamically adjust frequency
```

```c
void NCO_AdjustFrequency(NCO_Config *nco) {
    nco->frequency += nco->freq_step;

    // Clamp the frequency to stay within bounds
    if (nco->frequency > nco->max_freq) {
        nco->frequency = nco->max_freq;
    } else if (nco->frequency < nco->min_freq) {
        nco->frequency = nco->min_freq;
    }

    NCO_SetFrequency(nco, nco->frequency); // Apply the new frequency
}


// Function to read wave output
int16_t NCO_ReadWaveOutput(NCO_Config *nco) {
    int32_t wave_output = NCO_IP_mReadReg(nco->base_addr, REG_WAVE_OUTPUT_OFFSET);
    return wave_output & 0xFFFF; // Return lower 16 bits
}


int main(void) {
  int i;
    xil_printf("-- Inicio de NCO IP Core --\r\n");



    NCO_Config nco;
    NCO_Init(&nco, NCO_BASE_ADDR, 0x00100000, 0x00010000, 0x00010000, 0xFFFFFFFF);

    NCO_SetFrequency(&nco, nco.frequency);
    NCO_SetWaveType(&nco, nco.wave_type);
    NCO_SetEnable(&nco, nco.enable);

    while(1) {

        NCO_AdjustFrequency(&nco);

        // Read and display wave output
        int16_t wave_output = NCO_ReadWaveOutput(&nco);
        xil_printf("Wave output: %d\r\n", wave_output);
        for (i=0; i<9999999;i++);
    }

    NCO_SetEnable(&nco, 0x00);

    xil_printf("-- Fin de NCO IP Core --\r\n");
    return 0;
}
```

Listing 7: NCO$_interact$

La funcion del codigo anterior es interactuar con la NCO y generar distintas frecuencias en un rango establecido para lograr visualizarlas desde el ILA.

1.3. Diagrama de bloques

El diagrama de bloques de la NCO es presentado mediante el esquematico generado por Vivado.

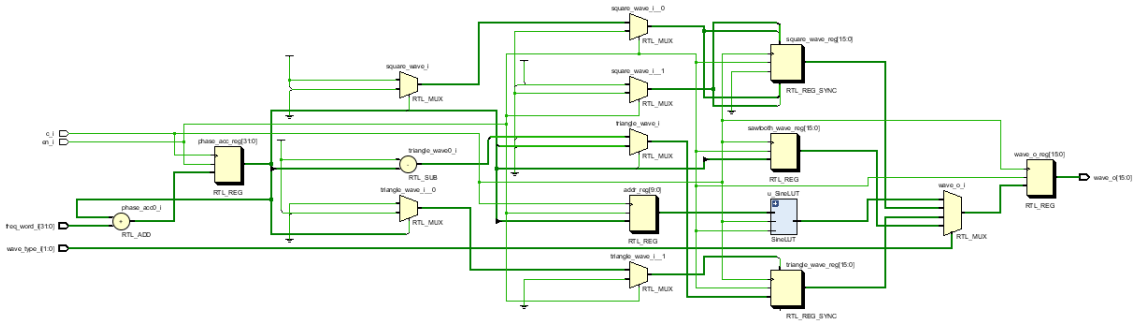

Figura 5: Diagrama de bloques NCO.

El diagrama del sineLUT es el siguiente:

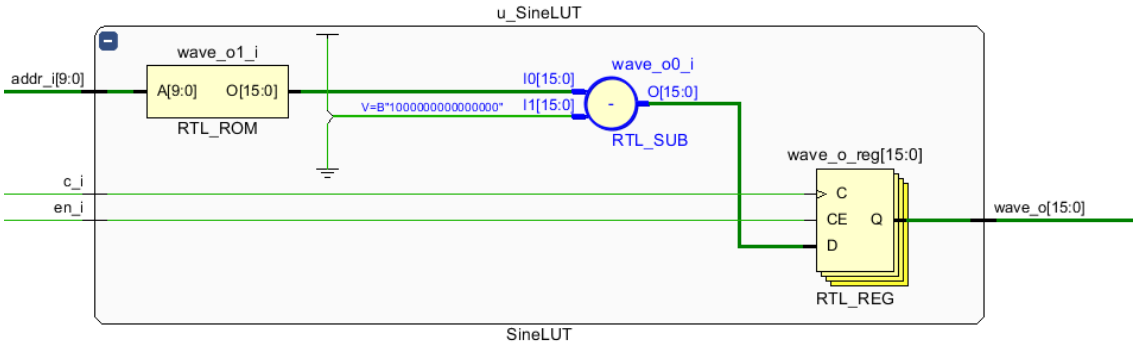

Figura 6: Diagrama de bloques sineLUT.

El diagrama de la NCO integrado con ZYNQ es el siguiente:

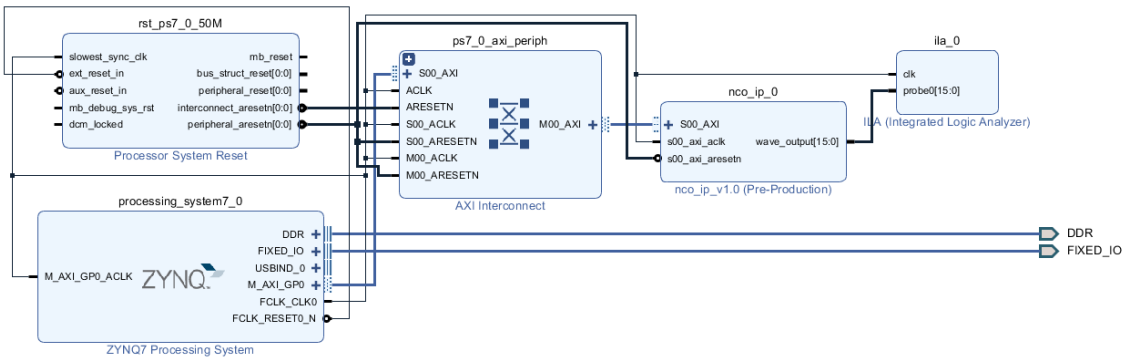

Figura 7: Diagrama de bloques de integracion.

2. Simulaciones y Pruebas

Utilizando Vivado y un testbench se simularon distintas formas de onda mostradas en la imagen siguiente:

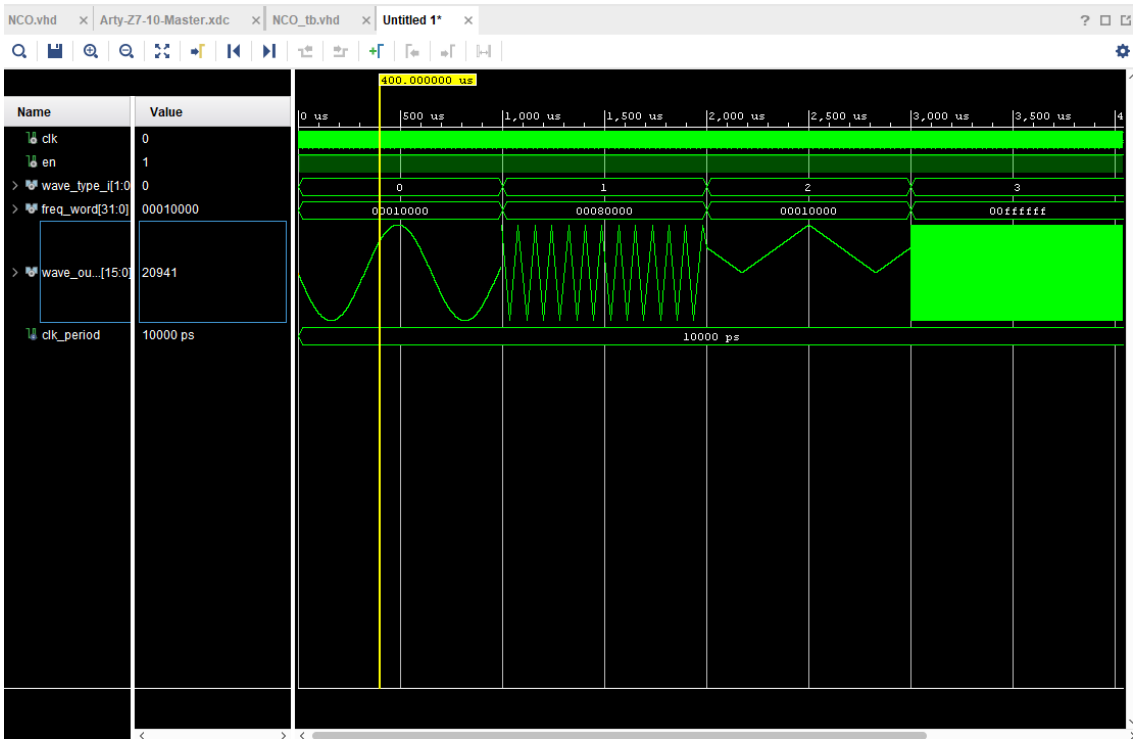

Figura 8: Simulaciones

El testbench cuenta con estímulos para generar ondas a distintas frecuencias:

```
1
2  -- Stimulus process to apply different test cases
3   stimulus : process
4   begin
5     -- Apply different wave types and frequency tuning words
6     wave_type_i <= "00"; -- Sine wave
7     freq_word   <= x"00010000"; -- Low frequency
8     wait for 1000 us;
9
10    wave_type_i <= "01"; -- Square wave
11    freq_word   <= x"00080000"; -- Medium frequency
12    wait for 1000 us;
13
14    wave_type_i <= "10"; -- Triangle wave
15    freq_word   <= x"00010000"; -- High frequency
16    wait for 1000 us;
17
18    wave_type_i <= "11"; -- Sawtooth wave
19    freq_word   <= x"00FFFFFF"; -- Higher frequency
20    wait for 1000 us;
```

Listing 8: SineLUT

Mediante el codigo C y la ILa se realizaron pruebas con la funcion seno:

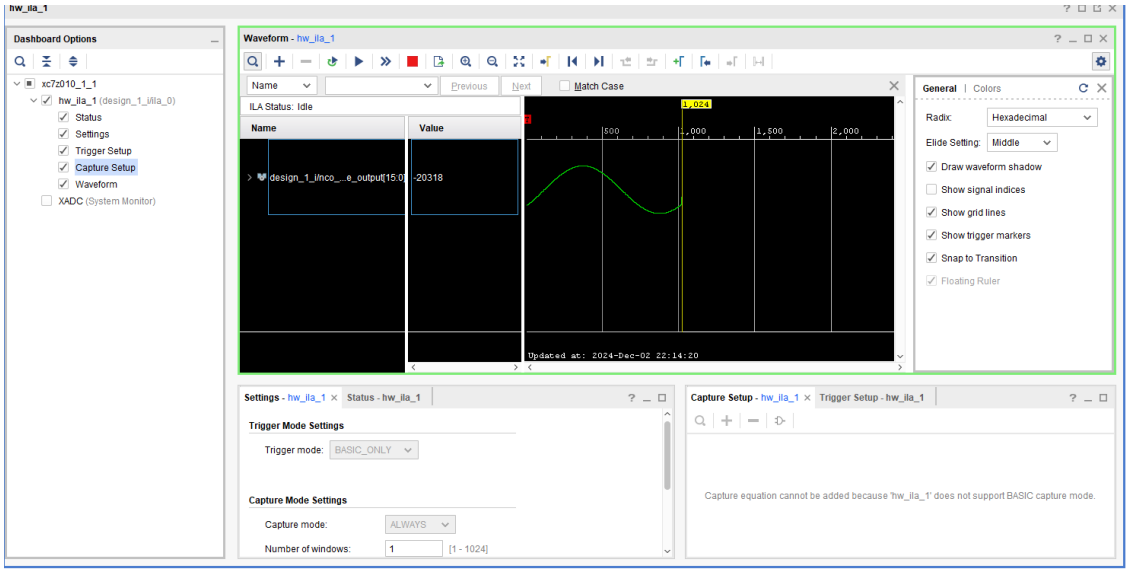

Figura 9: Prueba I mediante C.

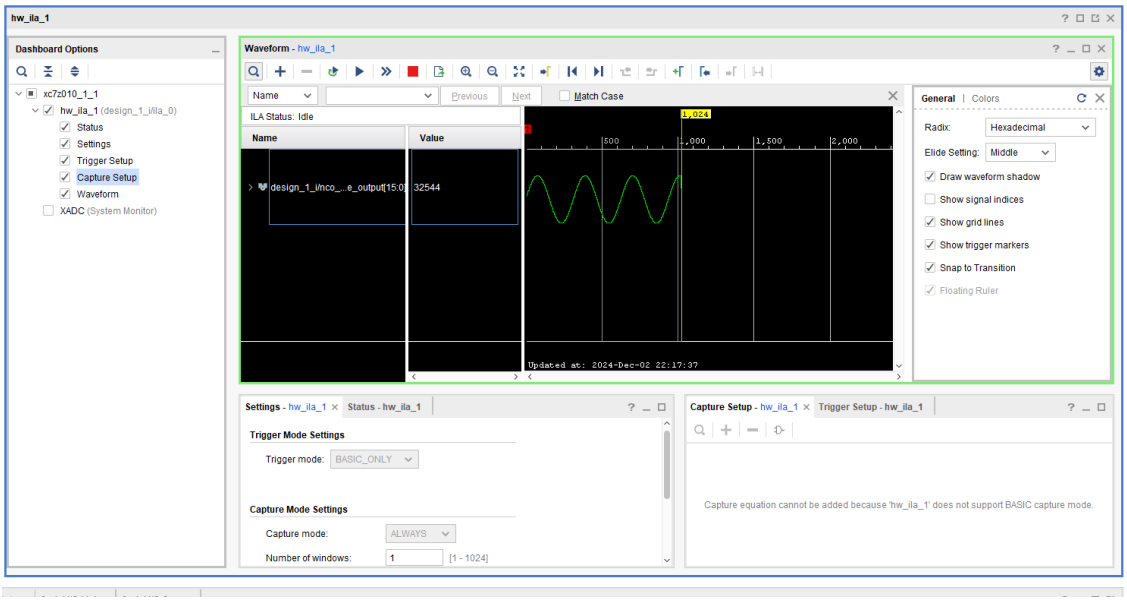

Figura 10: Prueba II mediante C.

Se probaron los outputs del NCO como registro, imprimiendo los valores por UART, aunque esto solo funciona a bajas frecuencias. Con los valores obtenidos por UART, se reconstruyó la onda usando Python.

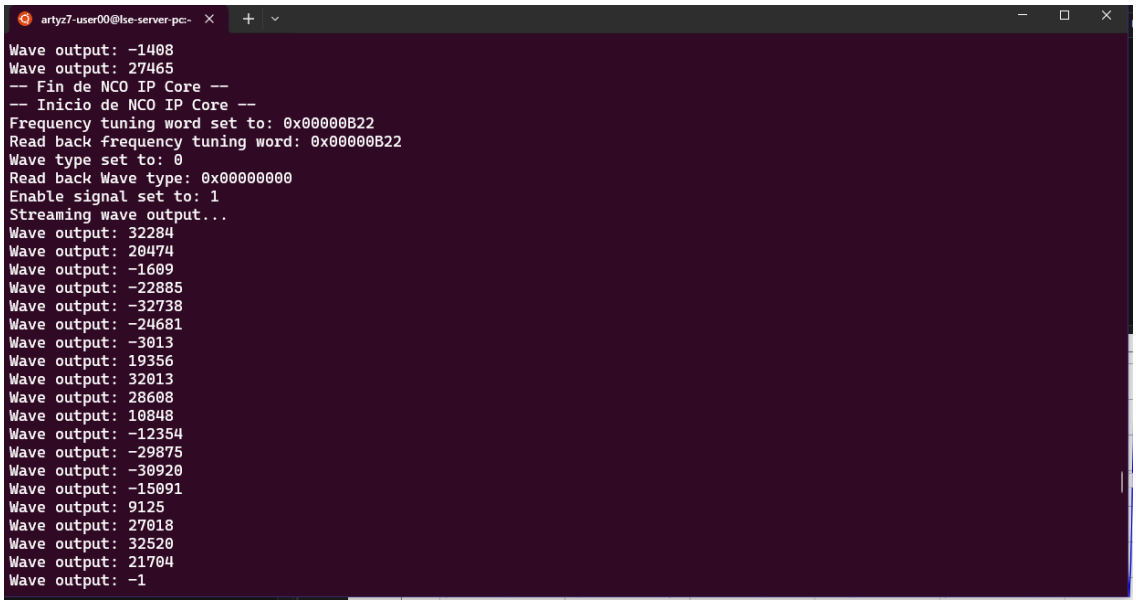

Figura 11: Valores del seno con UART.

La onda presentada mediante UART es una funcion seno de 2.8Khz, la reconstruccion es la siguiente:

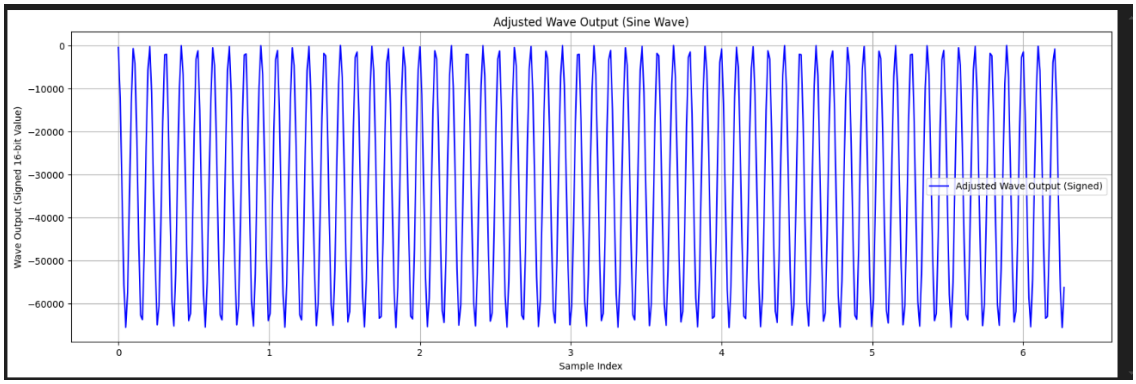

Figura 12: Reconstrucción de valores UART.

3. Tabla de recursos

La tabla de recursos generada por Vivado se demuestra en la siguiente imagen:

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	1506	17600	8.56
LUTRAM	156	6000	2.60
FF	2474	35200	7.03
BRAM	1	60	1.67
BUFG	2	32	6.25

Figura 13: Tabla de recursos.