Università di Pisa

NCO: Numerically Controlled Oscillator

Martina Troscia

21/04/2016

Contents

Introduction

A numerically controlled oscillator is a digital signal generator which creates a synchronous, discrete-time, discrete-valued representation of a waveform, usually sinusoidal.

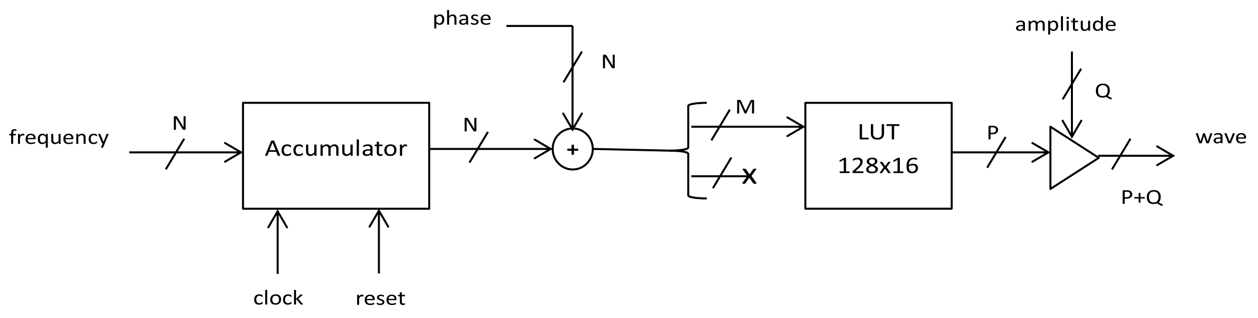
NCOs are used in many communication systems including digital up/down converters used in 3G wireless and software radio systems, radar systems, drivers for optical or acoustic transmissions and multilevel FSK/PSK modulators and demodulators.

A NCO generally consists of two parts:

- A phase accumulator, which adds to the value held at its output a frequency control value at each clock sample.
- A phase-to-amplitude converter, which uses the phase accumulator output word as an index into a waveform look-up table to provide a corresponding amplitude sample.

When clocked, the phase accumulator creates a saw tooth waveform, which is then converted by the phase-to-amplitude converter to a sample-domain waveform. The phase output word is usually truncated, as is it used to access the read-only memory containing the samples of the desired waveform which typically is a sinusoid. However, the truncation does not affect the frequency accuracy but produces a time-varying periodic phase error. In addition, various tricks are employed to further reduce the amount of memory required by the LUT. This include various trigonometric expansions, trigonometric approximations and, above all, methods which take advantage of the quadrature symmetry exhibited by sinusoids.

Architecture and implementation



This diagram shows the architecture of the implemented NCO.

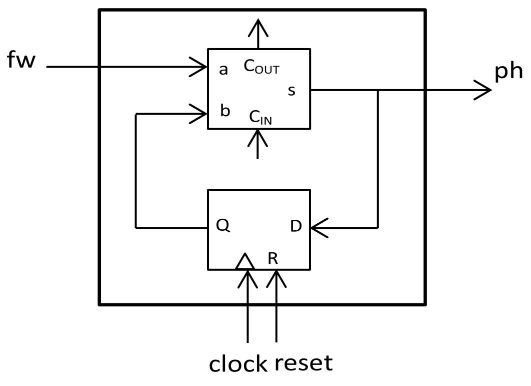
The device has 3 inputs, frequency and phase on N=16 bits and amplitude on Q=8 bits; in addition, it is given a pin to be reset and another one to receive the clock signal. The device emits a sinusoidal signal, represented on P+Q=24 bits.

At each clock cycle, the frequency value is “accumulated” and the obtained value is added to the phase value in order to delay the waveform and to obtain the correct index for accessing the LUT. The waveform emitted by the LUT is then amplified by using a multiplier.

Now the several components making up the NCO are presented before seeing the implementation of the device.

Accumulator

The accumulator is implemented as a register and an adder connected such that the output of the former is given as input to the latter and vice versa.



```
-------------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity accumulator is
       generic (N : INTEGER:=4);
       port(
              fw     : in  std_ulogic_vector(N-1 downto 0);
              clk    : in  std_ulogic;
              reset  : in  std_ulogic;
              ph     : out std_ulogic_vector(N-1 downto 0)
       );
end accumulator;

architecture acc_arch of accumulator is

       component adder
              generic (N : INTEGER:=4);
              port(
                     a      : in  std_ulogic_VECTOR (N-1 downto 0);
                     b      : in  std_ulogic_VECTOR (N-1 downto 0);
                     cin    : in  std_ulogic;
                     s      : out std_ulogic_VECTOR (N-1 downto 0);
                     cout   : out std_ulogic
              );
       end component adder;

       component reg
              generic (N: integer:=4);
              port(
                     d      : in  std_ulogic_vector(N-1 downto 0);
                     clk    : in  std_ulogic;
                     reset  : in  std_ulogic;
                     q      : out std_ulogic_vector(N-1 downto 0)
              );
       end component reg;

       signal reg_to_add: std_ulogic_vector(N-1 downto 0);
       signal add_to_reg: std_ulogic_vector(N-1 downto 0);


       begin
              Iadd:
                     adder generic map (N=>N)
                            port map (a=>fw, b=>reg_to_add, cin=>'0',
                                          s=>add_to_reg, cout=>open);
              Ireg:
                     reg generic map (N=>N)
                            port map (d=>add_to_reg, clk=>clk, reset=>reset, q=>reg_to_add);
              ph<=add_to_reg;

end acc_arch;
---------------------------------------------------------------------------------------
```

Adder and register implementations are showed in more detail below.

Adder

```
---------------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity adder is
   generic (N : INTEGER:=4);
   port(     a         : in  std_ulogic_VECTOR (N-1 downto 0);
             b         : in  std_ulogic_VECTOR (N-1 downto 0);
             cin       : in  std_ulogic;
             s         : out std_ulogic_VECTOR (N-1 downto 0);
          cout         : out std_ulogic
```

```
       );
end adder;

architecture add_arch of adder is

begin
    sum:
              process(a, b, cin)
                     variable C: std_ulogic;
                     begin
                            C := cin;
                            for i in 0 to N-1 loop
                                   s(i) <= a(i) XOR b(i) XOR C;
                                   C := (a(i) AND b(i)) OR (a(i) AND C) OR (b(i) AND C);
                            end loop;
                            cout <= C;
       end process sum;

end add_arch;
-----------------------------------------------------------------------------------------------
```

Register

The N-bit register is composed by N D-edged-triggered flip-flops that sample the value presented to their input at the rising edge of the clock. It is the only clocked component into the NCO design. Its content can be reset by emitting "0" on the reset pin (it is an active-low signal).

```
-----------------------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity dff is
       port
       (      d      : in  std_logic;
              clk    : in  std_logic;
              reset  : in  std_logic;
              q      : out std_logic
       );
end dff;

architecture dff_arch of dff is

begin
    idff:
              process(clk)
                     begin
                     if (clk'EVENT AND clk='1') then
                            q <= reset AND d;
                     end if;
    end process idff;

end dff_arch;
-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity reg is
       generic (N: integer:=4);
       port(d        : in  std_ulogic_vector(N-1 downto 0);
            clk      : in  std_ulogic;
            reset    : in  std_ulogic;
            q        : out std_ulogic_vector(N-1 downto 0)
       );
end reg;
```

```
architecture reg_arch of reg is
component dff

		port (
			clk	: in std_ulogic;
			reset	: in std_ulogic;
			d	: in std_ulogic;
			q	: out std_ulogic
		);
	end component dff;

	begin
		Ireg:
			for i in 0 to N-1 generate
				i_ff : dff
					port map (clk=>clk, reset=>reset, d=>d(i), q=>q(i));
			end generate Ireg;
end reg_arch;
---------------------------------------------------------------------------------------------
```

LUT

The accumulator produces a periodic ramp with a slope proportional to the input; this means that the $[0, 2\pi)$ phase interval is mapped into the $[0, 2^M - 1]$ interval at different frequencies. In other words, the angle is quantized on M bits, producing

$$\Delta\theta_q \in [0, 2^M - 1] * LSB_\theta$$

with

$$LSB_\theta = \frac{2\pi}{2^M}$$

It is also necessary to quantize the amplitude of the sinusoidal waveform $y = \sin\left(k\frac{2\pi}{2^M}\right)$, $k \in [0, 2^M - 1]$, produced as output. First of all, it is shifted by 1 so that only positive values have to be represented. Then, it is scaled by the factor

$$LSB_A = \frac{2}{2^P - 1}$$

with P being the number of bits used to represent the output and a quantized waveform amplitude is produced

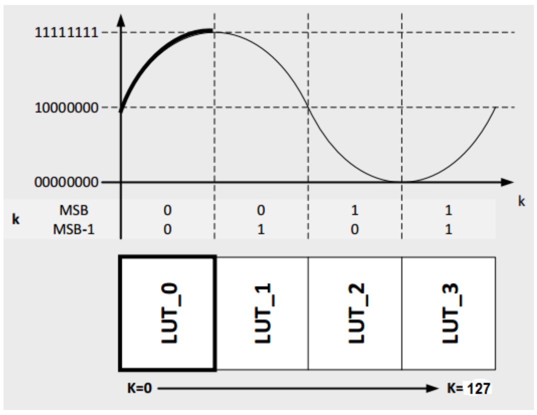
$$y_q(k) = \frac{1 + \sin\left(k\frac{2\pi}{2^M}\right)}{LSB_A}$$

All these operations were done by using the following Matlab function. It requires the desired number of bits for the input and the output, M and P respectively, and prints the result into a file called *lut_un_sin_MxP.*

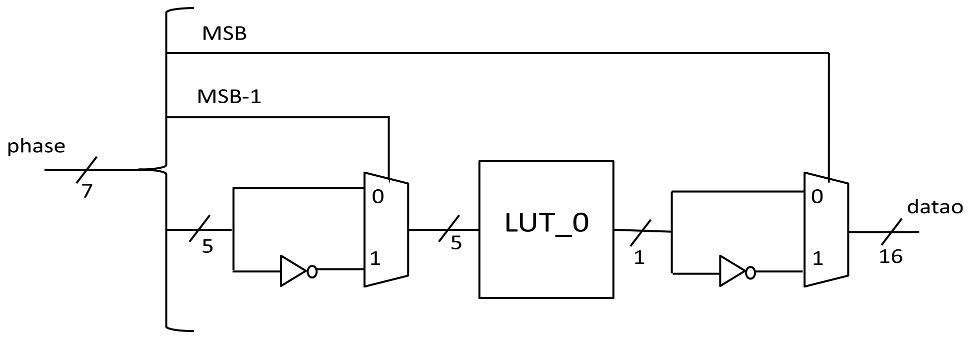
```
---------------------------------------------------------------------------------------------
function [] = lut_un_sin_MxP(M, P)

	for i=1:2^M
		k(i) = i-1;
	end

	in = sin(k*2*pi/2^M);
	LSB = 2/(2^P-1);
	out = round((1+in)/LSB);
```

```
        out_bin = dec2bin(out, P);

        name = ['lut_un_sin_' int2str(2^M) 'x' int2str(P) '.txt'];
        fileID = fopen(name, 'w');


        for i=1:2^M
                fprintf(fileID, '%d\t =>\t "%s"', k(i), out_bin(i, :));
                if (i~=2^M)
                        fprintf(fileID, ',\n');
                end
        end
        fclose(fileID);
end
---------------------------------------------------------------------------------
```

Since sinusoidal waves have a symmetrical profile, only the first quart of the LUT may be used to generate the same result. In this way, it is necessary a smaller ROM to store the *sin* values.



Every time the phase accumulator produces a value, it is not entirely used to access the LUT. *MSB-1 bit* is used to determine if the LUT has to be accessed from the top or from the bottom; *MSB bit* determines if data has to be complemented or not. The remaining bits address the smaller LUT. The following diagram represents this implementation.



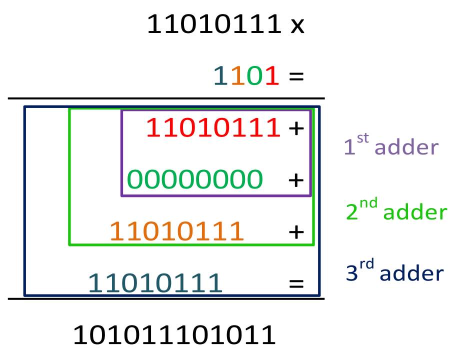
This is the VHDL code implementing the multiplexer and the LUT.

```
---------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity multiplexer is
        generic ( N: integer := 4);
        port
        (
                sel:    in std_ulogic;
                opt1:  in std_ulogic_vector(N-1 downto 0);
                opt2:  in std_ulogic_vector(N-1 downto 0);
                output:out std_ulogic_vector(N-1 downto 0)
        );
end multiplexer;

architecture multiplexer_arch of multiplexer is
        begin
                output <= opt1 when (sel='1') else opt2;
end multiplexer_arch;
------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity LUT_UN_SIN_32x16 is
        port (
                phase : in  std_ulogic_vector(6 downto 0);
                datao : out std_ulogic_vector(15 downto 0)
        );
end LUT_UN_SIN_32x16;

architecture lut_arch of LUT_UN_SIN_32x16 is

        component multiplexer is
                generic (
                        N: integer := 4
                );

                port (
                        sel    : in std_ulogic;
                        opt1   : in std_ulogic_vector(N-1 downto 0);
                        opt2   : in std_ulogic_vector(N-1 downto 0);
                        output : out std_ulogic_vector(N-1 downto 0)
                );
        end component multiplexer;

        type LUT_t is array (natural range 0 to 31) of std_ulogic_vector(15 downto 0);
        constant LUT: LUT_t := (
                0       =>      "1000000000000000",
                1       =>      "1000011001000111",
                2       =>      "1000110010001011",
                3       =>      "1001001011000111",
                4       =>      "1001100011111000",
                5       =>      "1001111100011001",
                6       =>      "1010010100100111",
                7       =>      "1010101100011111",
                8       =>      "1011000011111011",
                9       =>      "1011011010111001",
                10      =>      "1011110001010110",
                11      =>      "1100000111001101",
                12      =>      "1100011100011100",
                13      =>      "1100110000111111",
                14      =>      "1101000100110011",
                15      =>      "1101010111110101",
                16      =>      "1101101010000010",
                17      =>      "1101111011010111",
                18      =>      "1110001011110001",
                19      =>      "1110011011001111",
                20      =>      "1110101001101101",
                21      =>      "1110110111001001",
                22      =>      "1111000011100010",
                23      =>      "1111001110110101",
                24      =>      "1111011001000001",
                25      =>      "1111100010000100",
                26      =>      "1111101001111100",
                27      =>      "1111110000101001",
                28      =>      "1111110110001001",
                29      =>      "1111111010011100",
                30      =>      "1111111101100001",
                31      =>      "1111111111011000"
        );
```

```
	signal notphase	: std_ulogic_vector(4 downto 0);
	signal outmul1	: std_ulogic_vector(4 downto 0);
	signal outmul1_u	: unsigned (4 downto 0);
	signal inmul2	: std_ulogic_vector(15 downto 0);
	signal notinmul2	: std_ulogic_vector(15 downto 0);
	signal outmul2	: std_ulogic_vector(15 downto 0);

	begin

		notphase <= not phase(4 downto 0);

		Imul1:
			multiplexer generic map (N=>5)
				port map (sel => phase(5), opt1 => notphase,
					opt2 => phase(4 downto 0), output => outmul1);

		outmul1_u <= unsigned(outmul1);
		inmul2 <= LUT(to_integer(outmul1_u));

		notinmul2 <= not inmul2;
		Imul2:
			multiplexer generic map (N=>16)
				port map (sel => phase(6), opt1 => notinmul2,
					opt2 => inmul2, output => outmul2);

		datao <= outmul2;

end lut_arch;
--------------------------------------------------------------------------------------------
```

Multiplier

The sinusoidal waveform needs to be amplified: this is done by using a multiplier. It is built in a structural manner, reproducing the algorithm used when computing the multiplication by hand: each bit of the second operand is multiplied for the first one and the partial products are summed after being shifted of one position at every new line to obtain the final result. This means that there must be P Mx1 multipliers, with P being the number of bits of the second operand, and the results they produce must be summed by using P-1 adders.



This can be seen in the code below.

```
--------------------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity simple_mul is
```

```
        generic (M : INTEGER:=4);


        port(
                        a       : in std_ulogic_vector (M-1 downto 0);
                        bi      : in std_ulogic;

                        prod    : out std_ulogic_vector (M-1 downto 0)
        );
end simple_mul;

architecture simple_mul_arch of simple_mul is
        begin
                imul:
                process(a, bi)
                        begin
                                for i in 0 to M-1 loop
                                        prod(i) <= a(i) AND bi;
                                end loop;
         end process imul;
end simple_mul_arch;
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity multiplier is

        generic (
                M: integer:=16;
                P: integer:=8
        );

        port (
                a       : in std_ulogic_vector (M-1 downto 0);
                b       : in std_ulogic_vector (P-1 downto 0);
                prod    : out std_ulogic_vector (M+P-1 downto 0)
        );
end multiplier;


architecture mul_arch of multiplier is

        component simple_mul is
                generic (M : INTEGER:=4);
                port(
                                a       : in std_ulogic_vector (M-1 downto 0);
                                bi      : in std_ulogic;

                                prod: out std_ulogic_vector (M-1 downto 0)
                );
        end component simple_mul;


        component adder is

        generic (N : INTEGER:=4);
        port(
                a       : in  std_ulogic_VECTOR (N-1 downto 0);
                b       : in  std_ulogic_VECTOR (N-1 downto 0);
                cin     : in  std_ulogic;
                s       : out std_ulogic_VECTOR (N-1 downto 0);
                cout    : out std_ulogic);
        end component adder;

        --signals to support outputs of simple_muls
        type outsmul_t is array (natural range 0 to P-1) of std_ulogic_vector(M-1 downto 0);
        signal outsmul : outsmul_t;

        --signals to create inputs for adders
        type inadd_t is array (natural range 0 to P-2) of std_ulogic_vector(M-1 downto 0);
        signal inadd : inadd_t;
```

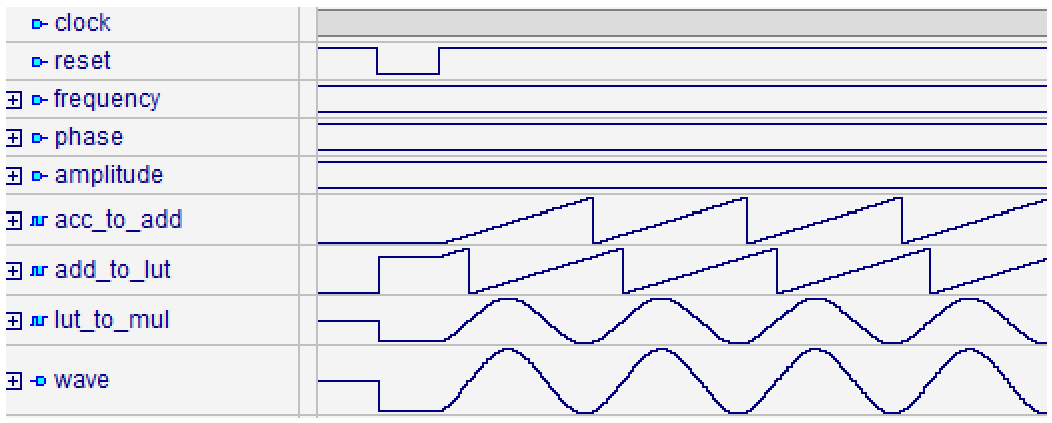
```
	--signals to support outputs of adders
	type outadd_t is array (natural range 0 to P-2) of std_ulogic_vector(M-1 downto 0);
	signal outadd : outadd_t;

	--signals to support carry_outs of adders
	type coutadd_t is array (natural range 0 to P-2) of std_ulogic;
	signal coutadd : coutadd_t;

	begin
		--partial products computation
		mulvec:
			for i in 0 to P-1 generate
				smul: simple_mul generic map (M) port map (a, b(i), outsmul(i));
			end generate mulvec;

		--link to adders the partial products to generate the output
			inadd(0) <= '0' & outsmul(0)(M-1 downto 1);
		add_wires:
			for i in 1 to P-2 generate
				inadd(i) <= coutadd(i-1) & outadd(i-1)(M-1 downto 1);
			end generate add_wires;

		addvec:
			for i in 0 to P-2 generate
		sadd: adder generic map (M) port map (inadd(i), outsmul(i+1), '0',
								outadd(i), coutadd(i));
			end generate addvec;

		--link the adder outputs to the one of the multiplier
		prod(0) <= outsmul(0)(0);
		mul_wires:
			for i in 1 to P-2 generate
				prod(i) <= outadd(i-1)(0);
			end generate mul_wires;
		prod(M+P-2 downto P-1) <= outadd(P-2)(M-1 downto 0);
		prod(M+P-1) <= coutadd(P-2);

end mul_arch;
-----------------------------------------------------------------------------------------------
```

NCO

The following VHDL code is the implementation of the NCO.

```
-----------------------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity nco is
	generic (
		--bits for input
		N : INTEGER:= 16;
		--bits for signal entering the LUT
		M : INTEGER:= 7;
		--bits for signal coming out from the LUT
		P : INTEGER:= 16;
		--bits for amplitude signal
		Q : INTEGER:= 8
	);
	port (
		clock		: in std_ulogic;
		reset		: in std_ulogic;

		frequency	: in std_ulogic_vector(N-1 downto 0);
		phase		: in std_ulogic_vector(N-1 downto 0);
		amplitude	: in std_ulogic_vector(Q-1 downto 0);

		wave		: out std_ulogic_vector(P+Q-1 downto 0)
  );
end nco;


architecture nco_arch of nco is

       component accumulator is
              generic (N : INTEGER:=4);
              port(
                     fw      : in  std_ulogic_vector(N-1 downto 0);
                     clk     : in  std_ulogic;
                     reset   : in  std_ulogic;
                     ph      : out std_ulogic_vector(N-1 downto 0)
              );
       end component accumulator;

       component adder is
              generic (N : INTEGER:=4);
              port(
                     a       : in  std_ulogic_VECTOR (N-1 downto 0);
                     b       : in  std_ulogic_VECTOR (N-1 downto 0);
                     cin     : in  std_ulogic;
                     s       : out std_ulogic_VECTOR (N-1 downto 0);
                     cout    : out std_ulogic
              );
       end component adder;

       component LUT_UN_SIN_32x16 is
              port (
                     phase   : in  std_ulogic_vector(6 downto 0);
                     datao   : out std_ulogic_vector(15 downto 0)
              );
       end component LUT_UN_SIN_32x16;

       component multiplier is
              generic (
                     M: integer:=8;
                     P: integer:=8
              );
              port (
                     a              : in std_ulogic_vector (M-1 downto 0);
                     b              : in std_ulogic_vector (P-1 downto 0);
                     prod           : out std_ulogic_vector (M+P-1 downto 0)
              );
       end component multiplier;

       --utility signals
       signal acc_to_add: std_ulogic_vector(N-1 downto 0);
       signal add_to_lut: std_ulogic_vector(N-1 downto 0);
       signal lut_to_mul: std_ulogic_vector(P-1 downto 0);

       begin

              Iacc: accumulator generic map (N=>N)
                     port map (fw=>frequency, clk=>clock, reset=>reset, ph=>acc_to_add);

              Iadd: adder generic map (N=>N)
                     port map (a=>phase, b=>acc_to_add, cin=>'0', s=>add_to_lut, cout=>open);


              Ilut: LUT_UN_SIN_32x16
                     port map (phase=>add_to_lut(N-1 downto N-7), datao=>lut_to_mul);

              Imul: multiplier generic map (M=>Q, P=>P)
                     port map (a=>amplitude, b=>lut_to_mul, prod=>wave);

end nco_arch;
--------------------------------------------------------------------------------
```

The following graph shows the behavior of the device when certain frequency, phase and amplitude values are presented to its input. The initial reset is necessary to start generating consistent values.



It is evident how the frequency input is quantized into a saw tooth waveform, is shifted by the quantity specified by the phase input and becomes a quantized sinusoid after exiting the LUT. The final amplification can be not entirely appreciated by the graph as it is not evident that the second wave is represented on a higher number of bits.

Verification

Testbench

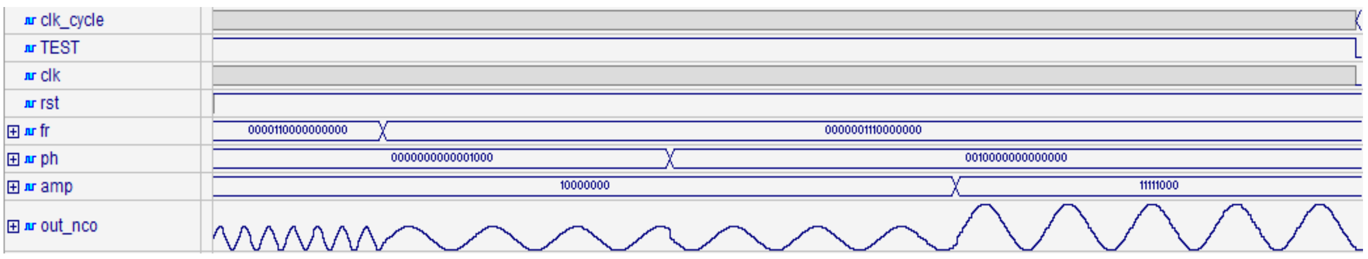
A simple test plan was defined to test the behavior of the device. The input values chosen at the beginning of the simulation are changed one at time to show how the output is affected by the input signals.

The “built” clock signal has a period of 200ns and a duty cycle of 50%, and it is generated until the variable TEST is set to *false*. At every rising and falling edge of the clock a testing process is executed. This process testes the value of the variable COUNT and determines the timing steps at which other values must be presented to the device. When the counter reaches the value TEST_LEN-1, the variable TEST is set to false and the simulation finishes.

```
-----------------------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity ncotb is

end ncotb;

architecture ncotb_arch of ncotb is

	component nco is
		generic (
			--bits for input
			N : INTEGER:= 16;
			--bits for signal entering the LUT
			M : INTEGER:= 7;
			--bits for signal coming out from the LUT
			P : INTEGER:= 16;
			--bits for amplitude signal
			Q : INTEGER:= 8
		);
		port (
			clock		: in std_ulogic;
			reset		: in std_ulogic;

			frequency	: in std_ulogic_vector(N-1 downto 0);
			phase		: in std_ulogic_vector(N-1 downto 0);
			amplitude	: in std_ulogic_vector(Q-1 downto 0);

			wave		: out std_ulogic_vector(P+Q-1 downto 0)
		);
	end component nco;


	constant CLOCK_PERIOD		: time		:= 200 ns;
	constant TEST_LEN		: integer	:= 2000;	-- Count number (CLOCK_PERIOD/2)
	constant N			: integer	:= 16;
	constant M			: integer	:= 7;
	constant P			: integer	:= 16;
	constant Q			: integer	:= 8;

	-- INPUT SIGNALS
	signal clk		: std_ulogic := '0';
	signal rst		: std_ulogic := '1';
	signal fr		: std_ulogic_vector(N-1 downto 0)  := "0000110000000000";
	signal ph		: std_ulogic_vector(N-1 downto 0)  := "0000000000001000";
	signal amp		: std_ulogic_vector(Q-1 downto 0)  := "10000000";
```

```
       -- OUTPUT SIGNALS
       signal out_nco       : std_ulogic_vector(P+Q-1 downto 0);
       signal clk_cycle     : integer := 0;
       signal TEST          : boolean := true;

       begin

       I: nco generic map (N=>N, M=>M, P=>P, Q=>Q)
              port map(clock=>clk, reset=>rst, frequency=>fr, phase=>ph, amplitude=>amp,
                     wave=>out_nco);


  -- Generate clk
       clk <= not clk after CLOCK_PERIOD/2 when TEST else '0';

  -- Run simulation for TEST_LEN cycles
       test_proc: process(clk)
       variable COUNT: integer:= 0;

              begin
                     clk_cycle <= (COUNT+1)/2;
                     case COUNT is
                            when 1                  =>     rst <= '0';
                            when 3                  =>     rst <= '1';
                            when 300                =>     fr <= "0000001110000000";
                            when 800                =>     ph <= "0010000000000000";
                            when 1300               =>     amp <= "11111000";
                            when (TEST_LEN - 1)     =>     TEST <= false;
                            when others             =>     null;
              end case;
                     COUNT:= COUNT + 1;
       end process test_proc;

end ncotb_arch;
--------------------------------------------------------------------------------------------
```

This image shows the testbench output.



At the 150th clock cycle the frequency value is changed from 0000110000000000 to 0000001110000000. The output wave starts oscillating more slowly and this behavior is expected as the second frequency value is smaller. At the 400th clock cycle the phase value is set to 0010000000000000: in this way the wave is delayed, as expected. At the 650th clock cycle it is possible to appreciate how the wave's amplitude increases when a greater value of amplitude is presented as input to the NCO.

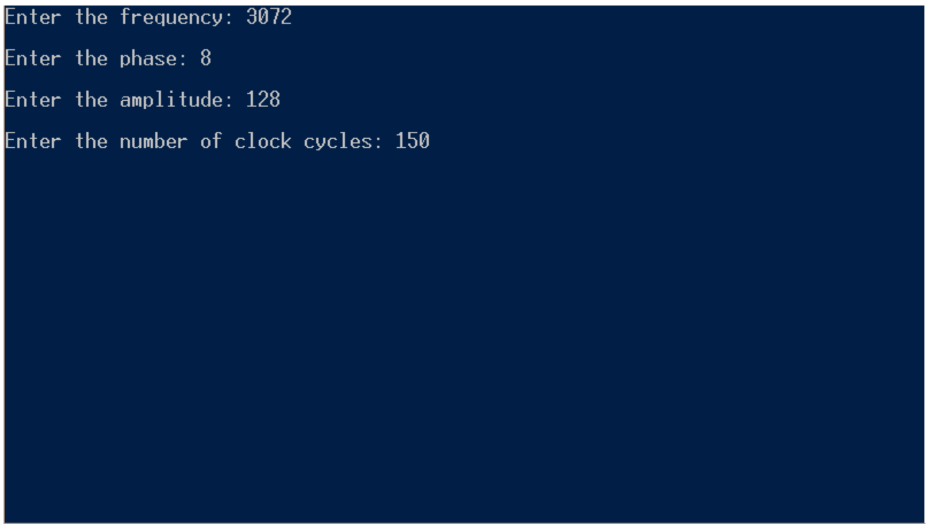
C++ program

A simple C++ program was made to test the correctness of the results emitted by the NCO. It requires the user to specify the frequency, phase and amplitude values and the number of clock cycles the device has to work. The program prints the value of the output at each clock cycle, without considering the first one where the device has just been reset and the output value is not a valid one.

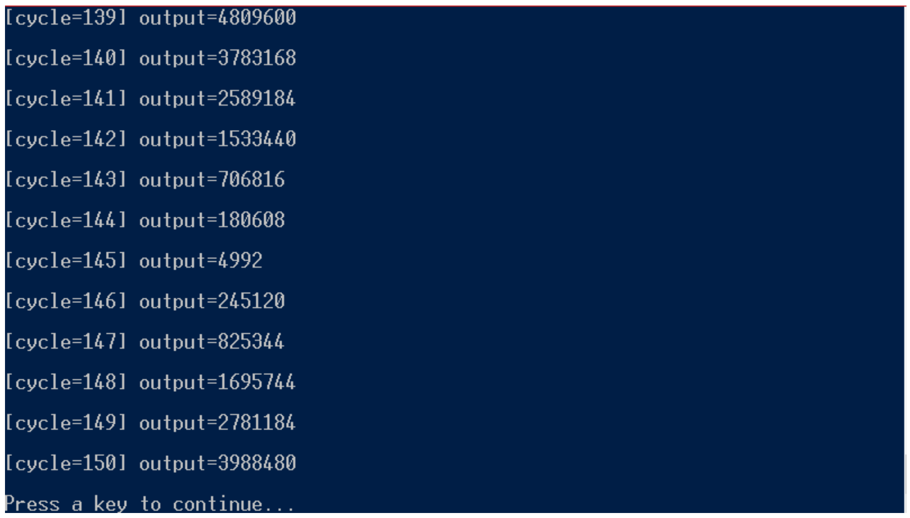
```
--------------------------------------------------------------------------------------------
#include<iostream>
#include<cstdlib>
#include<bitset>
#include<string>
using namespace std;

int main()
{
	unsigned short frequency, phase, amplitude;
	unsigned short num_cycles;
	unsigned short counter = 0;

	cout << "Enter the frequency: ";
	cin >> frequency;
	cout << endl << "Enter the phase: ";
	cin >> phase;
	cout << endl << "Enter the amplitude: ";
	cin >> amplitude;
	cout << endl << "Enter the number of clock cycles: ";
	cin >> num_cycles;

	for (int i=0; i<=num_cycles-2; i++)
	{
		counter += frequency;

		unsigned short inlut = counter+phase;

		std::bitset<16> inlut_bit(inlut);
		std::bitset<2> MSB;

		//set bits used for controls
		MSB[0]=inlut_bit[14];
		MSB[1]=inlut_bit[15];
		if (MSB[0]==1)
		{
			inlut_bit = ~inlut_bit;
		}

		std::bitset<5> inlut_bit_tr;
		for (int j=0; j<5; j++)
			inlut_bit_tr[j] = inlut_bit[9+j];
		int inlut_tr = inlut_bit_tr.to_ulong();

		unsigned short outlut;
		switch(inlut_tr)
		{
			case 0:
				outlut = 32768;
				break;
			case 1:
				outlut = 34375;
				break;
			case 2:
				outlut = 35979;
				break;
			case 3:
				outlut = 37575;
				break;
			case 4:
        outlut = 39160;
        break;
case 5:
        outlut = 40729;
        break;
case 6:
        outlut = 42279;
        break;

case 7:
        outlut = 43807;
        break;
case 8:
        outlut = 45307;
        break;
case 9:
        outlut = 46777;
        break;
case 10:
        outlut = 48214;
        break;
case 11:
        outlut = 49613;
        break;
case 12:
        outlut = 50972;
        break;
case 13:
        outlut = 52287;
        break;
case 14:
        outlut = 53555;
        break;
case 15:
        outlut = 54773;
        break;
case 16:
        outlut = 55938;
        break;
case 17:
        outlut = 57047;
        break;
case 18:
        outlut = 58097;
        break;
case 19:
        outlut = 59087;
        break;
case 20:
        outlut = 60013;
        break;
case 21:
        outlut = 60873;
        break;
case 22:
        outlut = 61666;
        break;
case 23:
        outlut = 62389;
        break;
case 24:
        outlut = 63041;
        break;
case 25:
        outlut = 63620;
        break;
case 26:
        outlut = 64124;
        break;
case 27:
        outlut = 64553;
        break;
case 28:
```

```
                        outlut = 64905;
                        break;
                case 29:
                        outlut = 65180;
                        break;
                case 30:
                        outlut = 65377;
                        break;

                case 31:
                        outlut = 65496;
                        break;
        }

        std::bitset<16> outlut_bit (outlut);
        if (MSB[1]==1)
                outlut_bit = ~outlut_bit;
        unsigned int out = outlut_bit.to_ulong();

        out *= amplitude;

        //print out the output value
        cout << endl << "[cycle=" << i+2 << "] output=" << out << endl;
    }

    char c;
    cout << endl << "Press a key to continue... ";
    cin >> c;
    return 0;
}
```

An example of execution is the following.

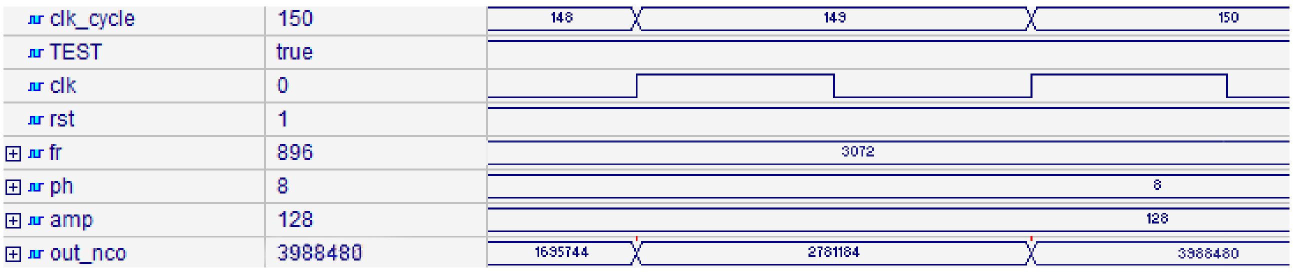


```
Enter the frequency: 3072

Enter the phase: 8

Enter the amplitude: 128

Enter the number of clock cycles: 150
```



```
[cycle=139] output=4809600
[cycle=140] output=3783168
[cycle=141] output=2589184
[cycle=142] output=1533440
[cycle=143] output=706816
[cycle=144] output=180608
[cycle=145] output=4992
[cycle=146] output=245120
[cycle=147] output=825344
[cycle=148] output=1695744
[cycle=149] output=2781184
[cycle=150] output=3988480
Press a key to continue...
```

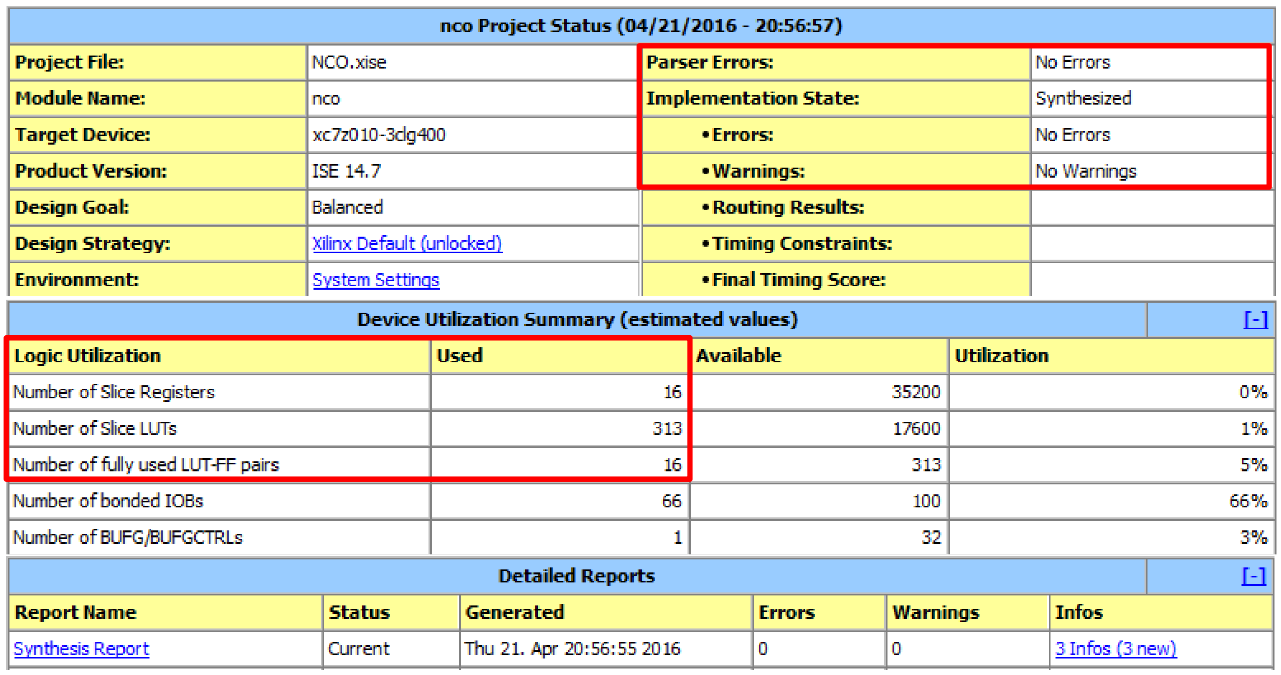
In this way the correctness of the result can be proved.



clk_cycle	150
TEST	true
clk	0
rst	1
fr	896
ph	8
amp	128
out_nco	3988480

Synthesis

The very last step is the synthesis of the project on Xilinx Zync by using the ISE tool. This is a summary of it.



nco Project Status (04/21/2016 - 20:56:57)			
Project File:	NCO.xise	**Parser Errors:**	No Errors
Module Name:	nco	**Implementation State:**	Synthesized
Target Device:	xc7z010-3clg400	**•Errors:**	No Errors
Product Version:	ISE 14.7	**•Warnings:**	No Warnings
Design Goal:	Balanced	**•Routing Results:**	
Design Strategy:	Xilinx Default (unlocked)	**•Timing Constraints:**	
Environment:	System Settings	**•Final Timing Score:**	

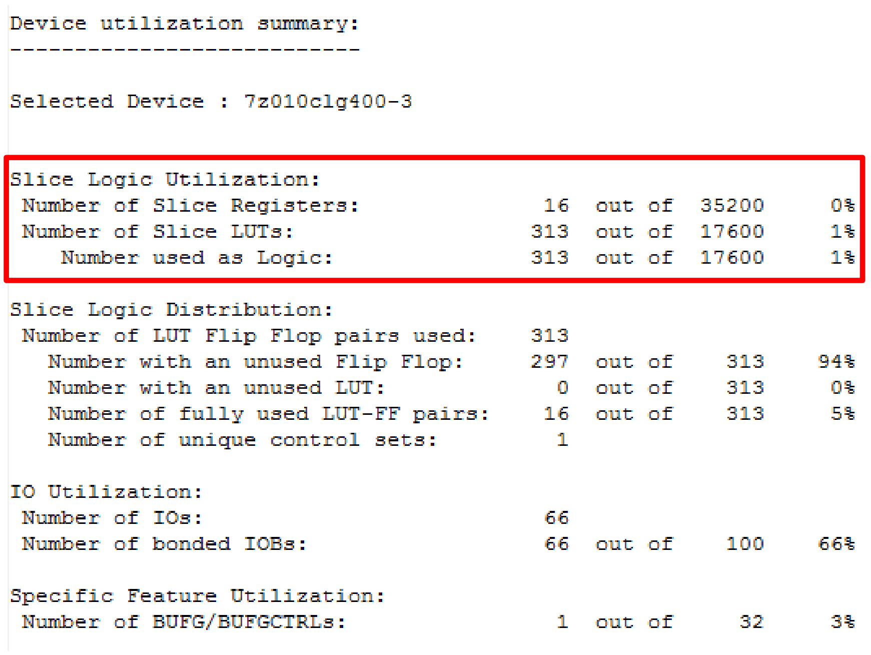
Device Utilization Summary (estimated values) [-]

Logic Utilization	Used	Available	Utilization
Number of Slice Registers	16	35200	0%
Number of Slice LUTs	313	17600	1%
Number of fully used LUT-FF pairs	16	313	5%
Number of bonded IOBs	66	100	66%
Number of BUFG/BUFGCTRLs	1	32	3%

Detailed Reports [-]

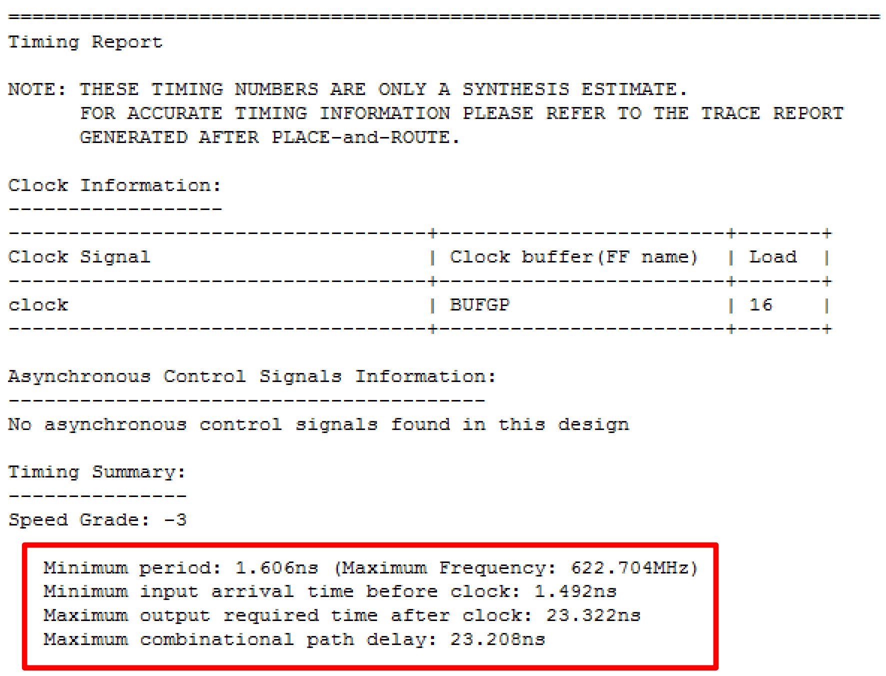
Report Name	Status	Generated	Errors	Warnings	Infos
Synthesis Report	Current	Thu 21. Apr 20:56:55 2016	0	0	3 Infos (3 new)

The following images are taken from the synthesis report. This first part of the report contains information related to the device utilization.



```
Device utilization summary:
---------------------------

Selected Device : 7z010clg400-3

Slice Logic Utilization:
 Number of Slice Registers:              16  out of  35200     0%
 Number of Slice LUTs:                  313  out of  17600     1%
    Number used as Logic:               313  out of  17600     1%

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:    313
   Number with an unused Flip Flop:     297  out of    313    94%
   Number with an unused LUT:             0  out of    313     0%
   Number of fully used LUT-FF pairs:    16  out of    313     5%
   Number of unique control sets:         1

IO Utilization:
 Number of IOs:                          66
 Number of bonded IOBs:                  66  out of    100    66%

Specific Feature Utilization:
 Number of BUFG/BUFGCTRLs:                1  out of     32     3%
```

Here there are some timing considerations.



```
=========================================================================
Timing Report

NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.
      FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT
      GENERATED AFTER PLACE-and-ROUTE.

Clock Information:
------------------
-----------------------------------+------------------------+-------+
Clock Signal                       | Clock buffer(FF name)  | Load  |
-----------------------------------+------------------------+-------+
clock                              | BUFGP                  | 16    |
-----------------------------------+------------------------+-------+

Asynchronous Control Signals Information:
----------------------------------------
No asynchronous control signals found in this design

Timing Summary:
---------------
Speed Grade: -3

   Minimum period: 1.606ns (Maximum Frequency: 622.704MHz)
   Minimum input arrival time before clock: 1.492ns
   Maximum output required time after clock: 23.322ns
   Maximum combinational path delay: 23.208ns
```

The device has been clocked with a frequency of 150MHz; however, it is evident now that the clock can be speeded up until 600MHz, if the board on which the design will be put allows it.