|  |
| --- |
| Università di Pisa |
| NCO: Numerically Controlled Oscillator |
|  |
|  |
| Martina Troscia |
| 21/04/2016 |

|  |
| --- |
|  |

Contents

[Introduction 2](#_Toc449080735)

[Architecture and implementation 3](#_Toc449080736)

[Accumulator 3](#_Toc449080737)

[Adder 4](#_Toc449080738)

[Register 5](#_Toc449080739)

[LUT 6](#_Toc449080740)

[Multiplier 9](#_Toc449080741)

[NCO 11](#_Toc449080742)

[Verification 14](#_Toc449080743)

[Testbench 14](#_Toc449080744)

[C++ program 16](#_Toc449080745)

[Synthesis 20](#_Toc449080746)

# 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

P+Q

P

Q

M

N

N

N

phase

frequency

x

**+**

Accumulator

LUT 128x16

clock

reset

amplitude

wave

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.

Q

D

R

b

s

CIN

COUT

a

fw

ph

clock

reset

---------------------------------------------------------------------------------------------

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 phase interval is mapped into the interval at different frequencies. In other words, the angle is quantized on M bits, producing

with

It is also necessary to quantize the amplitude of the sinusoidal waveform , , 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

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

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');

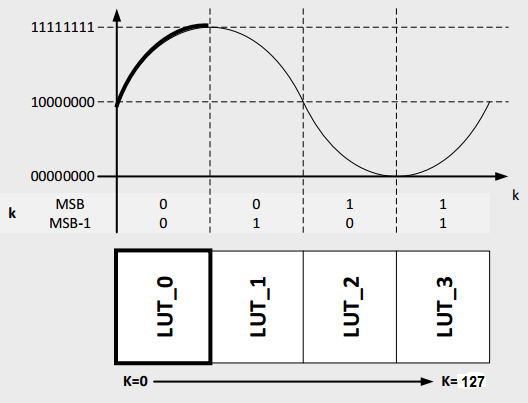
end

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.

5

MSB

MSB-1

16

16

5

7

LUT\_0

0

0

1

1

phase

datao

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.

11010111 x

1101 =

11010111 +

00000000 +

11010111 +

11010111 =

101011101011

1st adder

2nd adder

3rd adder

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

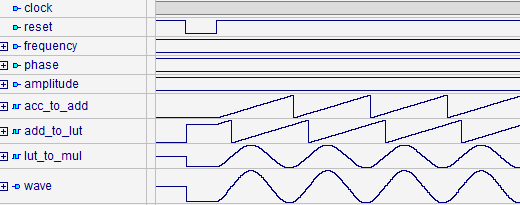
port map (phase=>add\_to\_lut(N-1 downto N-7), datao=>lut\_to\_mul);

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;

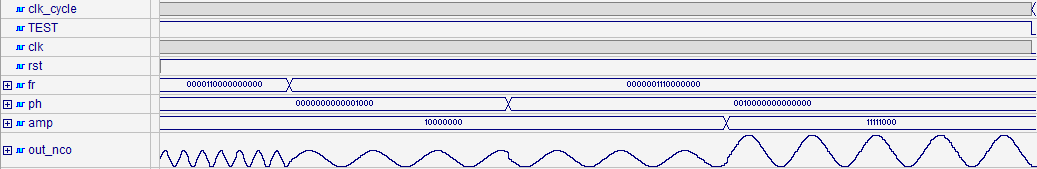
end case;

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;

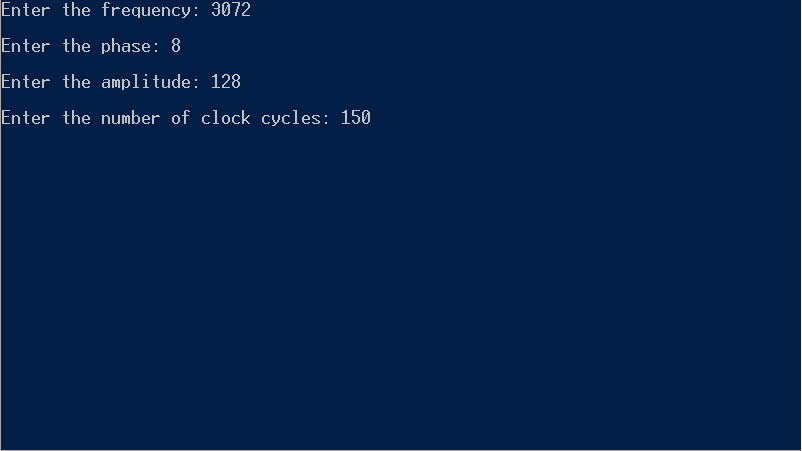
cout << endl << "Press a key to continue... ";

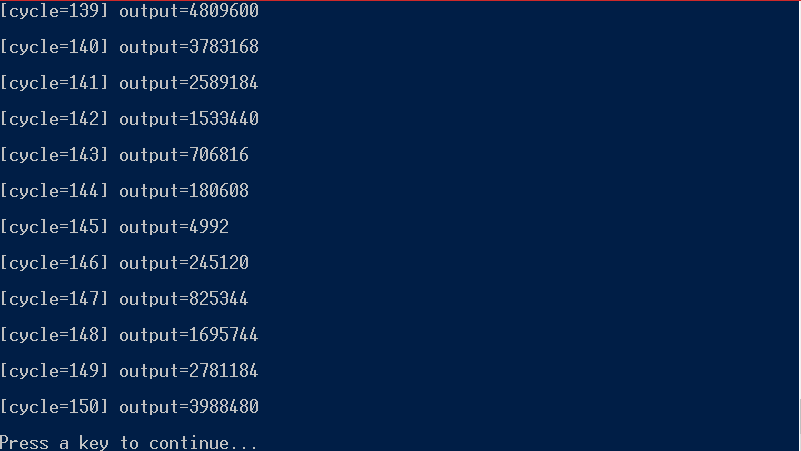
cin >> c;

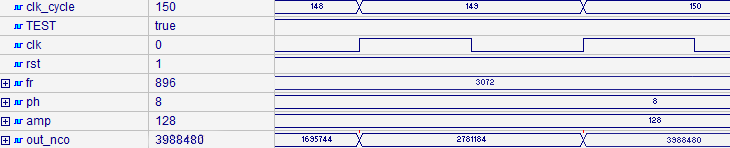
return 0;

}

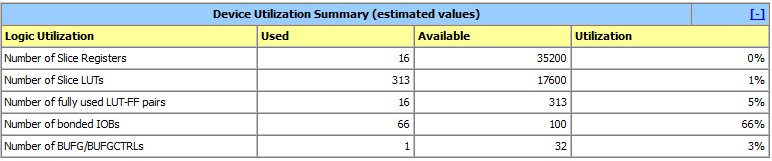
-----------------------------------------------------------------------------------------------

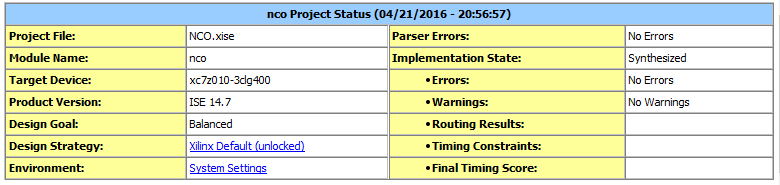
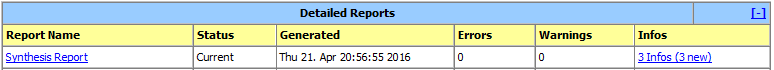
An example of execution is the following.

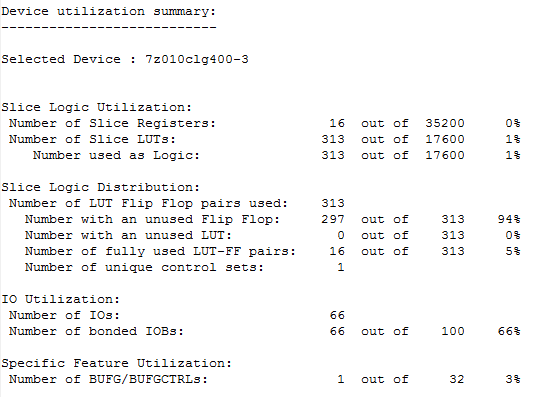
In this way the correctness of the result can be proved.

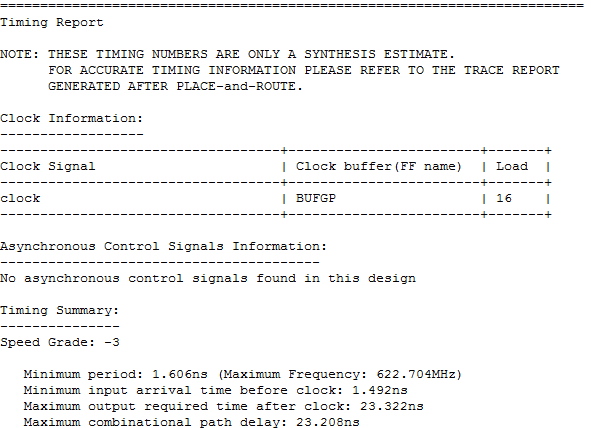


# Synthesis

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



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

Here there are some timing considerations.

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.