UNIVERSITY OF PISA

School of Engineering

Master of Science in Computer Engineering

Electronics and Communications Systems Course

PROJECT DOCUMENTATION

SINE FUNCTION APPROXIMATION USING INTERPOLATION

WORKGROUP:

Diego Casu

ACADEMIC YEAR 2019/2020

INDEX

1 INTRODUCTION

The sine function is one of the most important and recurring transcendental functions to be computed: real-time speed control, spatial positioning, digital waveform generation, computer graphics, wireless communications and signal processing represent only a small set of fields where it is widely employed. Given the heterogeneity of the application domains, it is fundamental to have at disposal a rich set of approximation methods able to evaluate the sine fulfilling different requirements for accuracy, speed and implementation costs. The latter are the discriminant that lead to adopt software or hardware solutions, exploiting programming languages on top of general-purpose processors or dedicated integrated circuits.

The objective of this project is to introduce a simple and fast electronic circuit for the computation of the sine function using interpolation, designed to work with inputs in fixed-point representation and to provide values with a good accuracy. The document is organized as follows: chapter 2 describes the most common approaches and algorithms for sine approximation; chapter 3 presents the design and architecture of the electronic circuit, together with its VHDL description; chapter 4 provides the relative test manual. Finally, chapter 5 reports the synthesis and implementation of the circuit for the Xilinx FPGA Zync platform, by means of the tool VIVADO.

2 METHODS FOR SINE APPROXIMATION

2.1 Taylor series

The Taylor series of a function $f(x)$ is an infinite sum of terms obtained from the function derivatives at a single point x_0, called center:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n$$

Imposing $x_0 = 0$ (Maclaurin series), it is possible to write the sine power series expansion:

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

where taking the first n terms means approximating the function as an n-degree polynomial.

This approach has the benefit of being conceptually simple, adaptable in terms of accuracy by increasing or decreasing the number of terms in the sum and implementable using only adders and multipliers, eventually reusing in an incremental way the previously evaluated powers or factorials; the latter could be simplified by pre-computing and storing the values in a lookup table.

The method is not suggested when a high accuracy is required: the slow convergence of the series would result in polynomials of high degree, impacting the speed of the circuit and becoming demanding in terms of number of bits needed for the representation of the powers/factorials.

2.2 Lookup tables

This method is the easiest in terms of implementation, since it requires only a memory support, for instance a ROM: it consists in pre-computing the values of the sine, storing them with the required accuracy and representation, and retrieving them with a proper indexing strategy when needed.

The designer has complete freedom over the granularity and representation of the original values, but the size of the table strongly increases to the point of being too demanding when the accuracy to achieve is high. With respect to the Taylor series approach, the problem can be mitigated combining the lookup strategy with interpolating techniques (linear or parabolic) and the reuse of values, due to the sine being a periodic odd function.

2.3 Chebyshev polynomials

A Chebyshev polynomial of degree n, denoted as $T_n(x)$, is defined as $T_n(x) = \cos(n\, cos^{-1}(x))$. Although looking trigonometric, explicit expressions for them exist and are:

$$T_0(x) = 1$$

$$T_1(x) = x$$

$$T_2(x) = 2x^2 - 1$$

...

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x) \quad n \geq 1$$

A Chebyshev polynomial $T_n(x)$ is such that in the interval $[-1, 1]$:

1) it has n zeros, located at $x = \cos\left(\frac{\pi\left(k-\frac{1}{2}\right)}{n}\right) \;\; k = 1, 2, \dots, n$;
2) it has $n + 1$ extrema, located at $x = \cos\left(\frac{\pi k}{n}\right) \;\; k = 1, 2, \dots, n$, such that $T_n(x) = 1$ at all of the maxima and $T_n(x) = -1$ at all of the minima.

These properties are the foundations of the approximation method based on Chebyshev series: given an arbitrary function $f(x)$ in $[-1, 1]$, call c_j the coefficients:

$$c_j = \frac{2}{N}\sum_{k=1}^{N} f\left[\cos\left(\frac{\pi\left(k-\frac{1}{2}\right)}{N}\right)\right]\cos\left(\frac{\pi j\left(k-\frac{1}{2}\right)}{N}\right) \qquad j = 0, 1, \dots, N-1$$

Then the approximation formula:

$$f(x) = \sum_{j=0}^{N-1} c_j T_j(x) - \frac{1}{2}c_0$$

is exact for x equal to the N zeros of $T_N(x)$, hence it approximates $f(x)$ in the interval $[-1, 1]$.

It is possible to observe that the coefficients c_j decrease rapidly (exponentially) and that, truncating the approximation at $m \ll N$, the error is dominated by the term $c_m T_m(x)$, where $T_m(x) \in [-1, 1]$. This behaviour can be exploited to approximate the sine in an efficient and precise way; in particular:

1) the coefficients c_j can be pre-computed and stored inside a lookup table, since their values depend only on the function shape and not on the specific point x;
2) the degree m of the Chebyshev approximating polynomial can be decided based on the values of the coefficients c_j and the requirements;

3) the approximation can be evaluated computing $T_j(x)$ using the initial recurrent formula and accumulating the sums or using directly the Clenshaw's recurrence formula:

$$d_{m+1} = d_m = 0$$

$$d_j = 2xd_{j+1} - d_{j+2} + c_j \quad j = m-1, m-2, \dots, 1$$

$$f(x) = d_0 = xd_1 - d_2 + \frac{1}{2}c_0$$

In any case, it requires only additions and multiplications.

2.4 CORDIC algorithm

CORDIC (Coordinate Rotation Digital Computer) is a simple and efficient algorithm to calculate elementary functions like square roots, logarithms and trigonometric functions using rotations.

Suppose to want to compute $\sin(\phi)$, with ϕ an arbitrary angle: since there exists a point (x_{target}, y_{target}) on the unit circle such that $x_{target} = \cos(\phi)$ and $y_{target} = \sin(\phi)$, it is possible to start from the coordinates $(1, 0)$ and reach it with incremental rotations by an angle β. Constraining the problem in the first quadrant, the search of $\sin(\phi)$ consists in the following steps:

1) start from $(1, 0)$, set $\beta = 45°$ and rotate;
2) if $\phi \geq \beta$, then set $\beta = \beta + (45/2)°$, else set $\beta = \beta - (45/2)°$; then rotate. This means that the rotation is performed in the same direction if the desired angle is not reached yet or in the opposite direction if the angle was exceeded in the previous step;
3) repeat step 2 until convergence, each time halving the increment/decrement of the angle ($45/4$, $45/8$, $45/16$, etc.).

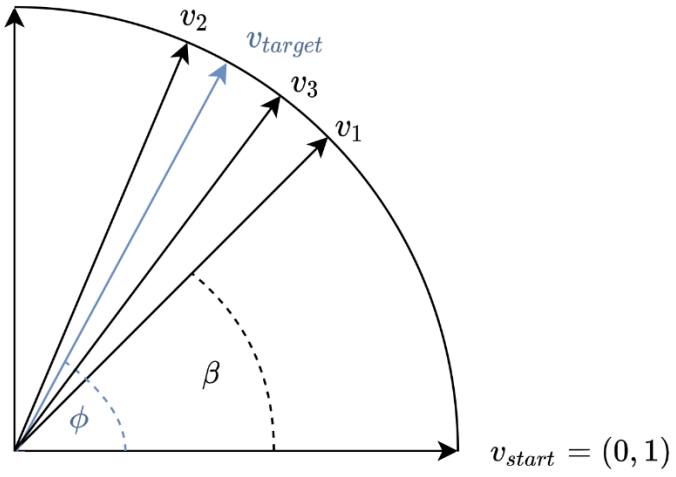

Figure 1 Update of the starting vector through rotations towards the target..

Given a point (x_i, y_i) at a step $i = 0, 1, \dots$, its rotation on the unit circle by an angle β is realized as:

$$x_{i+1} = x_i cos(\beta) - y_i d_i sin(\beta)$$
$$y_{i+1} = y_i cos(\beta) + x_i d_i sin(\beta)$$

where d_i is the rotate direction ($+1$ if the target angle is not reached, -1 if it was exceeded).

The expression can be rewritten as:

$$x_{i+1} = cos(\beta)[x_i - y_i d_i tan(\beta)]$$
$$y_{i+1} = cos(\beta)[y_i + x_i d_i tan(\beta)]$$

It is possible to observe that:

1) the multiplying term $cos(\beta)$ is a scaling factor and its overall impact through the iterations can be calculated in advance depending on the number of rotations to do, namely on the accuracy requirements. Hence, a single multiplication by the overall scaling factor can be performed at the beginning or at the end of the method, removing its need at each rotation;
2) the angle β can be chosen so that $tan(\beta) = 2^{-i}$, i.e. the tangent multiplication at each rotation can be reduced to a simple shift operation in hardware.

It follows that a CORDIC rotation reduces to the efficiently computable expression:

$$x_{i+1} = x_i - y_i d_i 2^{-i}$$
$$y_{i+1} = y_i + x_i d_i 2^{-i}$$

where d_i is computed as:

$$z_{i+1} = z_i - d_i \tan^{-1}(2^{-i}), \quad z_0 = \beta$$
$$d_i = \begin{cases} -1 & z_i < 0 \\ +1 & z_i \geq 0 \end{cases}$$

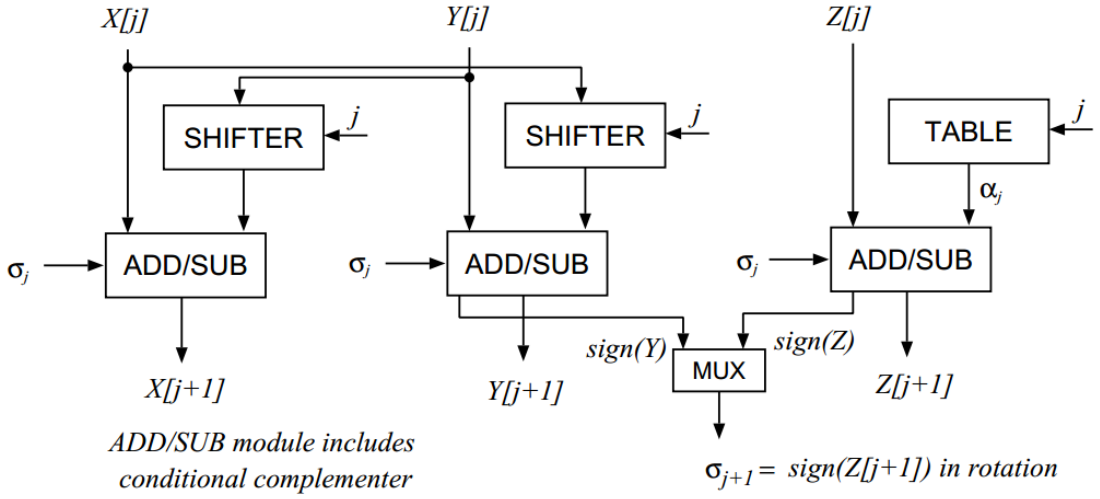

Figure 2 Implementation of one iteration of CORDIC algorithm (Ercegovac, M., e Lang T, Digital Arithmetic, 2003)

3 ELECTRONIC CIRCUIT FOR SINE APPROXIMATION

3.1 Architecture

The electronic circuit described in the following computes the sine function of a given value using a lookup table approach combined with linear interpolation: the result is a simple and fast design, with a good accuracy, but not suited for high precision applications.

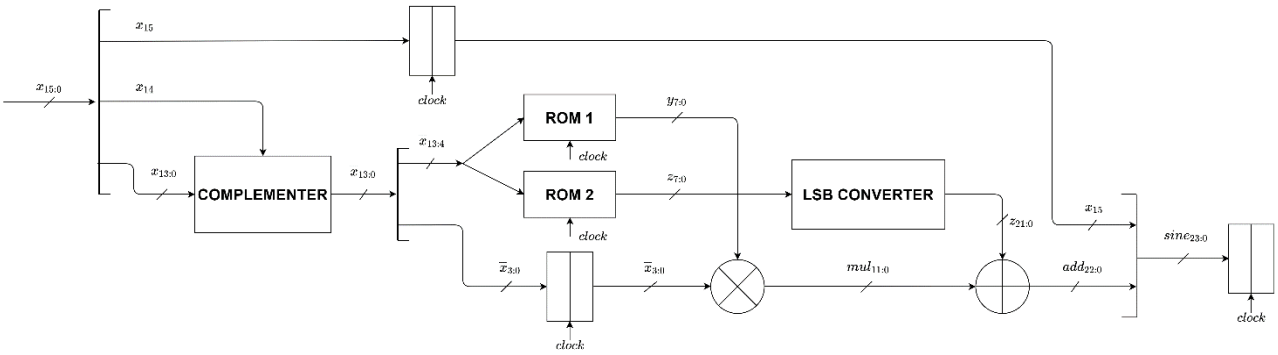

Figure 3 Electronic circuit for sine approximation using interpolation.

The circuit accepts 16-bit inputs in sign and magnitude representation defining radiant values constrained in the interval $[-\pi, \pi]$. The magnitude is expressed in fixed-point, where the LSB is:

$$LSB_{INPUT} = \frac{\pi}{2^{15} - 1}$$

Given the possible input values and the fact that the sine is odd, the sign of the input x_{15} is also the sign of the output and can be saved in a dedicated flip-flop; the remaining 15 bits are processed by a complement unit, which performs the bit-wise complement of $x_{13:0}$ if the enable bit x_{14} is set, namely shifting the phases in the interval $[\frac{\pi}{2}, \pi]$ to the interval $[0, \frac{\pi}{2}]$, reducing the number of pre-computed values to store in the lookup tables.

The complemented value $\bar{x}_{13:0}$ is split in two parts, $\bar{x}_{13:4}$ and $\bar{x}_{3:0}$, used for different purposes:

1) $\bar{x}_{13:4}$, interpreted as an unsigned, is used to address the two 1024x8 ROMs, while interpreted in fixed-point on 14 bits, i.e. with the bits 3:0 equal to 0, is used to pre-compute the stored values used for the interpolation. Called x_i the unsigned value and $\hat{x}_i$ the same value interpreted in fixed-point as said before, ROM1 stores $\frac{\sin(\hat{x}_{i+1}) - \sin(\hat{x}_i)}{\hat{x}_{i+1} - \hat{x}_i}$, while ROM2 stores $\sin(\hat{x}_i)$. Both the memories keep fixed-point numbers in the interval $[0, 1]$ on 8 bits, with an LSB equal to:

$$LSB_{ROM} = \frac{1}{2^8 - 1}$$

2) $\bar{x}_{3:0}$, which represents the value $\bar{x}_{13:0} - \bar{x}_{13:4}$, is saved in a dedicated flip-flop and then multiplied by the output of ROM1.

The output of the multiplier $mul_{11:0}$, with LSB equal to $LSB_{INPUT} \cdot LSB_{ROM}$ and the output of ROM2 $z_{7:0}$, with LSB equal to LSB_{ROM}, must be summed: since the sum of two numbers in fixed-point requires the inputs to use the same LSB, an LSB converter unit is introduced to represent $z_{7:0}$ with $LSB_{INPUT} \cdot LSB_{ROM}$. The choice is to convert $z_{7:0}$ and not $mul_{11:0}$: given that an LSB conversion is performed multiplying by the ratio $\frac{LSB_{START}}{LSB_{END}}$ and rounding the result to an integer, the conversion of $mul_{11:0}$ would be done multiplying by $\frac{LSB_{INPUT}\, LSB_{ROM}}{LSB_{ROM}} = LSB_{INPUT} \ll 1$, producing a rounding always equal to 0. The conversion of $z_{7:0}$ is done multiplying by $\frac{LSB_{ROM}}{LSB_{INPUT}\, LSB_{ROM}} = \frac{1}{LSB_{INPUT}} \gg 1$, with the downside of introducing a representation on 22 bits.

Finally, the converted $z_{7:0}$ and $mul_{11:0}$ are summed: the output of the adder $add_{22:0}$ is concatenated with the initial sign bit, composing the final output $sine_{23:0}$ on 24 bits in sign and magnitude representation, where the magnitude is expressed in fixed-point with an LSB equal to $LSB_{INPUT}\, LSB_{ROM}$.

3.2 VHDL description

The circuit can be described using the VHDL language: in the following, the definition of the elementary units is presented, with the final topology obtained linking them as shown in Figure 3.

3.2.1 D flip-flop

```
1. library IEEE;
2. use IEEE.std_logic_1164.all;
3.
4. entity DFlipFlop is
5.     generic (Nbit : positive); -- Number of bits to store in the flip-flop
6.     port (
7.         reset : in std_logic;  -- Active high asynchronous reset
8.         clock : in std_logic;
9.         input : in std_logic_vector (Nbit - 1 downto 0);
10.        output : out std_logic_vector (Nbit - 1 downto 0)
11.    );
```

```
12. end DFlipFlop;
13.
14. architecture Behaviour of DFlipFlop is
15. begin
16.
17.     dff_process: process(reset, clock)
18.     begin
19.         if reset = '1' then
20.             output <= (others => '0');
21.         elsif rising_edge(clock) then
22.             output <= input;
23.         end if;
24.     end process;
25.
26. end Behaviour;
```

The component is declared with a generic number of bits, since it is used in various portions of the circuit to save different sized operands, and it is characterized by an active high asynchronous reset.

3.2.2 Complementer

```
1.  library IEEE;
2.  use IEEE.std_logic_1164.all;
3.
4.  entity Complementer is
5.      generic (Nbit : positive);  -- Number of bits in input
6.      port (
7.          enable : in std_logic;  -- Active high enable signal
8.          input : in std_logic_vector (Nbit - 1 downto 0);
9.          output : out std_logic_vector (Nbit - 1 downto 0)
10.     );
11. end Complementer;
12.
13. architecture Behaviour of Complementer is
14. begin
15.
16.     compl_process: process(enable, input)
17.     begin
18.         if enable = '1' then
19.         -- Bit-wise complement of the input if enable is active
20.             output <= not(input);
21.         else
22.             output <= input;
23.         end if;
24.     end process;
25.
26. end Behaviour;
```

The complementer takes one input on *N* bits and returns its bit-wise complement on the same number of bits. The conversion is done only if the active high enable signal is set and it is performed with the help of the library function *not()*.

3.2.3 Multiplier

```
1. library IEEE;
2. use IEEE.std_logic_1164.all;
3. use IEEE.numeric_std.all;
4.
5. entity Multiplier is
6.     generic (
7.         N_bit_input_x : positive;
8.         N_bit_input_y : positive
9.     );
10.    port (
11.        x : in std_logic_vector (N_bit_input_x - 1 downto 0);
12.        y : in std_logic_vector (N_bit_input_y - 1 downto 0);
13.        output : out std_logic_vector ((N_bit_input_x + N_bit_input_y - 1)
14.                                                              downto 0)
15.    );
16.end Multiplier;
17.
18.architecture Behaviour of Multiplier is
19.begin
20.
21.    output <= std_logic_vector(unsigned(x)*unsigned(y));
22.
23.end Behaviour;
```

The multiplier accepts two inputs on different number of bits N and M, and returns their multiplication on $N + M$ bits. The operation is done interpreting the two inputs as unsigned, since the unit is designed to work on fixed-point values.

3.2.4 Adder

```
1. library IEEE;
2. use IEEE.std_logic_1164.all;
3. use IEEE.numeric_std.all;
4.
5. entity Adder is
6.     generic (Nbit : positive);
7.     port (
8.         x : in std_logic_vector (Nbit - 1 downto 0);
9.         y : in std_logic_vector (Nbit - 1 downto 0);
10.        output : out std_logic_vector (Nbit downto 0)
11.    );
12.end Adder;
13.
14.architecture Behaviour of Adder is
15.begin
16.
17.    output <= std_logic_vector(unsigned('0' & x) + unsigned('0' & y));
18.
19.end Behaviour;
```

The unit accepts two inputs on the same number of bits N and returns their sum on $N + 1$ bits. The inputs are positive fixed-point numbers, meaning that the operation can be done extending the inputs adding zeros as most significant bits and treating the operands as unsigned.

3.2.5 LSB converter

```
1. library IEEE;
2. use IEEE.std_logic_1164.all;
3. use IEEE.numeric_std.all;
4.
5. entity LSB_Converter is
6.     generic (Nbit : positive);  -- Number of bits of the input
7.     port (
8.         input : in std_logic_vector (Nbit - 1 downto 0);
9.         output : out std_logic_vector ((Nbit + 13) downto 0)
10.        -- Output on Nbit + 14 bits.
11.    );
12.end LSB_Converter;
13.
14.architecture Behaviour of LSB_converter is
15.
16.    -- Converting ratio LSB_start/LSB_end = 1/LSB_input = 10430.
17.    constant LSB_ratio : std_logic_vector (13 downto 0) := "10100010111110";
18.
19.begin
20.
21.    -- LSB conversion done scaling by a constant pre-established factor.
22.    output <= std_logic_vector(unsigned(input)*unsigned(LSB_ratio));
23.
24.end Behaviour;
```

The circuit converts a fixed-point number using an LSB equal to LSB_{ROM} to its equivalent using LSB_{INPUT} LSB_{ROM}. The conversion is done multiplying by the constant ratio $\frac{LSB_{ROM}}{LSB_{INPUT}\ LSB_{ROM}} = \frac{1}{LSB_{INPUT}} \cong 10430.062779$, which is rounded to 10430 for implementation purposes, so that the product becomes a simple integer operation between unsigned numbers, with a minimal loss in terms of accuracy.

3.2.6 ROMs

```
1. library IEEE;
2. use IEEE.std_logic_1164.all;
3. use IEEE.numeric_std.all;
4.
5. entity ROM_1 is
6.     port (
7.         reset : in std_logic;
8.         clock : in std_logic;
9.         address : in std_logic_vector (9 downto 0);
10.        data : out std_logic_vector (7 downto 0)
11.    );
12.end ROM_1;
13.
14.architecture Behaviour of ROM_1 is
15.
16.    -- 1024x8 ROM
17.    type rom_t is array (0 to 1023) of std_logic_vector (7 downto 0);
18.    constant rom : rom_t := (
19.                            "11111111",
20.                               ....
21.                            "00000001",
22.                            "00000000");
23.
24.begin
25.
26.    rom_proc: process(reset, clock)
27.    begin
28.        if reset = '1' then
29.            data <= (others => '0');
30.        elsif rising_edge(clock) then
31.            data <= rom(to_integer(unsigned(address)));
32.        end if;
33.    end process;
34.
35.end Behaviour;
```

```
1. library IEEE;
2. use IEEE.std_logic_1164.all;
3. use IEEE.numeric_std.all;
4.
5. entity ROM_2 is
6.     port (
7.         reset : in std_logic;
8.         clock : in std_logic;
9.         address : in std_logic_vector (9 downto 0);
10.        data : out std_logic_vector (7 downto 0)
11.    );
12.end ROM_2;
13.
14.architecture Behaviour of ROM_2 is
15.
16.    -- 1024x8 ROM
17.    type rom_t is array (0 to 1023) of std_logic_vector (7 downto 0);
```

```
18.     constant rom : rom_t := (
19.                               "00000000",
20.                                  ....
21.                               "11111111",
22.                               "11111111");
23.
24. begin
25.
26.     rom_proc: process(reset, clock)
27.     begin
28.         if reset = '1' then
29.             data <= (others => '0');
30.         elsif rising_edge(clock) then
31.             data <= rom(to_integer(unsigned(address)));
32.         end if;
33.     end process;
34.
35. end Behaviour;
```

The two 1024x8 ROMs store pre-computed values used for the interpolation, which are directly embedded inside the definition as constants. Since the sign bit and the less significant part of the complementer are stored inside flip-flops, both the ROMs output the addressed value synchronously with the clock, in order to avoid race conditions in the subsequent combinational networks; moreover, an active high asynchronous reset is provided, forcing the output to "00…0" when set. Globally, the ROMs behave like the combination of a lookup table and a flip-flop.

3.2.7 Sine interpolator

```
1.  library IEEE;
2.  use IEEE.std_logic_1164.all;
3.
4.  entity SineInterpolator is
5.      port (
6.          clock : in std_logic;
7.          reset : in std_logic;   -- Active high asynchronous reset
8.          input : in std_logic_vector (15 downto 0);
9.          output : out std_logic_vector (23 downto 0)
10.     );
11. end SineInterpolator;
12.
13. architecture Behaviour of SineInterpolator is
14.
15.     ----------- SIGNALS -----------
16.
17.     -- Output of the flip-flop saving the sign bit.
18.     -- It is a vector due to the general structure of the flip-flop.
19.     signal sign : std_logic_vector (0 downto 0);
20.
21.     -- Output of the complementer.
22.     signal out_complementer : std_logic_vector (13 downto 0);
23.
```

```
24.     -- Output of the ROMs.
25.     signal out_rom1 : std_logic_vector (7 downto 0);
26.     signal out_rom2 : std_logic_vector (7 downto 0);
27.
28.     -- Output of the flip-flop saving the less significant bits
29.     -- of the output of the complementer.
30.     -- They will be multiplied by the output of ROM 1.
31.     signal low_out_complementer : std_logic_vector (3 downto 0);
32.
33.     -- Output of the multiplier. The extended version is used as
34.     -- input of the adder.
35.     signal out_multiplier : std_logic_vector (11 downto 0);
36.     signal out_multiplier_extended : std_logic_vector (21 downto 0);
37.
38.     -- Output of the LSB converter.
39.     signal out_LSB_converter : std_logic_vector (21 downto 0);
40.
41.     -- Output of the adder.
42.     signal out_adder : std_logic_vector (22 downto 0);
43.
44.     -- Signal for the final result (combination of sign and out_adder).
45.     -- It will be the input of the flip-flop providing the output.
46.     signal sine : std_logic_vector (23 downto 0);
47.
48.     ----------- COMPONENTS -----------
49.
50.     component DFlipFlop is
51.     -- Number of bits to store in the flip-flop
52.         generic (Nbit : positive);
53.         port (
54.             reset : in std_logic;  -- Active high asynchronous reset
55.             clock : in std_logic;
56.             input : in std_logic_vector (Nbit - 1 downto 0);
57.             output : out std_logic_vector (Nbit - 1 downto 0)
58.         );
59.     end component;
60.
61.     component Complementer is
62.         generic (Nbit : positive);  -- Number of bits in input
63.         port (
64.             enable : in std_logic;  -- Active high enable signal
65.             input : in std_logic_vector (Nbit - 1 downto 0);
66.             output : out std_logic_vector (Nbit -1 downto 0)
67.         );
68.     end component;
69.
70.     component ROM_1 is
71.         port (
72.             reset : in std_logic;
73.             clock : in std_logic;
74.             address : in std_logic_vector (9 downto 0);
75.             data : out std_logic_vector (7 downto 0)
76.         );
77.     end component;
78.
79.     component ROM_2 is
80.         port (
81.             reset : in std_logic;
```

```
82.             clock : in std_logic;
83.             address : in std_logic_vector (9 downto 0);
84.             data : out std_logic_vector (7 downto 0)
85.         );
86.     end component;
87.
88.     component Multiplier is
89.         generic (
90.             N_bit_input_x : positive;
91.             N_bit_input_y : positive
92.         );
93.         port (
94.             x : in std_logic_vector (N_bit_input_x - 1 downto 0);
95.             y : in std_logic_vector (N_bit_input_y - 1 downto 0);
96.             output : out std_logic_vector (
97.                                 (N_bit_input_x + N_bit_input_y - 1) downto 0)
98.         );
99.     end component;
100.
101.        component LSB_Converter is
102.            generic (Nbit : positive);  -- Number of bits of the input
103.            port (
104.                input : in std_logic_vector (Nbit - 1 downto 0);
105.                output : out std_logic_vector ((Nbit + 13) downto 0)
106.                -- Output on Nbit + 14 bits.
107.            );
108.        end component;
109.
110.        component Adder is
111.            generic (Nbit : positive);
112.            port (
113.                x : in std_logic_vector (Nbit - 1 downto 0);
114.                y : in std_logic_vector (Nbit - 1 downto 0);
115.                output : out std_logic_vector (Nbit downto 0)
116.            );
117.        end component;
118.
119.
120.        ----------- WIRING AND BEHAVIOUR -----------
121.
122.    begin
123.
124.        -- Flip-flop saving the sign bit sampled directly from the input.
125.        sign_dff: DFlipFlop
126.            generic map (Nbit => 1)
127.            port map (
128.                reset => reset,
129.                clock => clock,
130.                input => input(15 downto 15),
131.                output => sign
132.            );
133.
134.        -- Complementer implementing the phase shifting from [pi/2, pi]
135.        -- to [0, pi/2], if enabled.
136.        compl: Complementer
137.            generic map (Nbit => 14)  -- Number of bits in input
138.            port map (
139.                enable => input(14),
```

```
140.                input => input(13 downto 0),
141.                output => out_complementer
142.            );
143.
144.        -- ROMs implementing the lookup tables storing the
145.        -- pre-computed values for the sine interpolation.
146.        -- Addressed by the 10 most significant bits of
147.        -- the output of the complementer.
148.        rom1: ROM_1
149.            port map (
150.                reset => reset,
151.                clock => clock,
152.                address => out_complementer(13 downto 4),
153.                data => out_rom1
154.            );
155.
156.        rom2: ROM_2
157.            port map (
158.                reset => reset,
159.                clock => clock,
160.                address => out_complementer(13 downto 4),
161.                data => out_rom2
162.            );
163.
164.        -- Flip-flop storing the 4 less significant bits of
165.        -- the output of the complementer.
166.        low_out_compl_dff: DFlipFlop
167.            generic map (Nbit => 4)
168.            port map (
169.                reset => reset,
170.                clock => clock,
171.                input => out_complementer(3 downto 0),
172.                output => low_out_complementer
173.            );
174.
175.        -- Multiplication between the output of ROM 1 and
176.        -- the less significant bits of the output of the complementer.
177.        mul: Multiplier
178.            generic map (
179.                N_bit_input_x => 4,
180.                N_bit_input_y => 8
181.            )
182.            port map (
183.                x => low_out_complementer,
184.                y => out_rom1,
185.                output => out_multiplier
186.            );
187.
188.        -- LSB conversion of the output of ROM 2 to use
189.        -- an LSB equal to LSB_INPUT*LSB_ROM.
190.        lsb_conv: LSB_Converter
191.            generic map (Nbit => 8)
192.            port map (
193.                input => out_rom2,
194.                output => out_lsb_converter
195.            );
```

```
196.        -- Sum of the output of the LSB converter, i.e. the output of ROM 2,
197.        -- and the output of the multiplier.
198.        add: Adder
199.            generic map (Nbit => 22)
200.            port map (
201.                x => out_multiplier_extended,
202.                y => out_lsb_converter,
203.                output => out_adder
204.            );
205.
206.        -- Flip-flop storing the final result.
207.        dff_output: DFlipFlop
208.            generic map (Nbit => 24)
209.            port map (
210.                reset => reset,
211.                clock => clock,
212.                input => sine,
213.                output => output
214.            );
215.
216.        -- Extension of the output of the multiplier to
217.        -- 22 bits from the starting 12 bits.
218.        out_multiplier_extended <= "0000000000" & out_multiplier;


219.        -- Merging of the sign and magnitude signals
220.        --to generate the sine signal.
221.        sine <= sign & out_adder;
222.
223.
224.    end Behaviour;
```

The circuit takes as input a number in sign and magnitude on 16 bits, where the latter is expressed in fixed-point, and returns an output on 24 bits with the same convention; it must be supplied with a clock and a reset. The behaviour of the unit is defined in an structural way, providing the components and the linking between them, as expressed in Figure 3, with the addition of a flip-flop to store the final output: an output not directly provided by a combinational network is useful during the synthesis/implementation phase to avoid a timing report with an infinite slack.

4 TEST MANUAL

4.1 Verification

The correctness of the design of the circuit has been verified comparing the output of the simulated VHDL code with an ad-hoc Python simulator. All the 2^{16} possible inputs have been tested: in particular, the outputs of the Python simulator have been pre-computed, stored in a dedicated ROM and compared at run-time with the ones provided by the hardware using a subtractor. The output of the latter has never changed from 0, meaning a perfect correspondence between the two implementations.

The testbench implementing the verification is the following, where `ROM_Test` and `Subtractor` are the components specifically introduced to perform the comparison:

```
1. library IEEE;
2. use IEEE.std_logic_1164.all;
3. use IEEE.numeric_std.all;
4.
5. entity SineInterpolator_tb is
6. end SineInterpolator_tb;
7.
8. architecture Test of SineInterpolator_tb is
9.
10.     constant T_CLOCK : time := 10 ns;
11.     constant T_RESET : time := 20 ns;
12.
13.     signal clock_tb : std_logic := '0';
14.     signal reset_tb : std_logic := '1';
15.     signal input_tb : std_logic_vector (15 downto 0);
16.     signal output_interpolator_tb : std_logic_vector (23 downto 0);
17.     signal output_rom_tb : std_logic_vector (23 downto 0);
18.     signal output_dff_tb : std_logic_vector (23 downto 0);
19.
20.     -- Difference between the output of the interpolator and
21.     -- the output of the ROM.
22.     -- Used to check if they are equal (all bits are 0) or not.
23.     signal difference_tb : std_logic_vector (24 downto 0);

24.     -- Active high signal to terminate the simulation
25.     signal end_sim : std_logic := '0';
26.
27.
28.     component SineInterpolator is
29.         port (
30.             clock : in std_logic;
31.             reset : in std_logic;   -- Active high asynchronous reset
32.             input : in std_logic_vector (15 downto 0);
```

```
            output : out std_logic_vector (23 downto 0)
        );
    end component;

    -- ROM used to store the outputs of the SineInterpolator
    -- simulated in Python.
    component ROM_Test is
        port (
            reset : in std_logic;
            clock : in std_logic;
            address : in std_logic_vector (15 downto 0);
            data : out std_logic_vector (23 downto 0)
        );
    end component;

    -- Flip-flop used to store the output of the ROM.
    -- It works as a delay component synchronizing the outputs of
    -- the interpolator and of the ROM for a correct comparation.
    component DFlipFlop is
        -- Number of bits to store in the flip-flop
        generic (Nbit : positive);
        port (
            reset : in std_logic;  -- Active high asynchronous reset
            clock : in std_logic;
            input : in std_logic_vector (Nbit - 1 downto 0);
            output : out std_logic_vector (Nbit - 1 downto 0)
        );
    end component;

    -- Compares the output of the interpolator and the output of the ROM.
    -- The subtractor is synchronous, so that the output is stable and its
    -- eventual transitions can be checked easily with the simulation tools.
    component Subtractor is
        generic (Nbit : positive);
        port (
            reset : in std_logic;
            clock : in std_logic;
            x : in std_logic_vector (Nbit - 1 downto 0);
            y : in std_logic_vector (Nbit - 1 downto 0);
            output : out std_logic_vector (Nbit downto 0)
        );
    end component;


begin

    reset_tb <= '0' after T_RESET;
    clock_tb <= (not(clock_tb) or end_sim) after T_CLOCK/2;

    sine_interpolator: SineInterpolator
        port map (
            clock => clock_tb,
            reset => reset_tb,
            input => input_tb,
            output => output_interpolator_tb
        );

    rom: ROM_Test
```

```
91.         port map (
92.             reset => reset_tb,
93.             clock => clock_tb,
94.             address => input_tb,
95.             data => output_rom_tb
96.         );
97.
98.     dff: DFlipFlop
99.         generic map (Nbit => 24)
100.           port map (
101.               reset => reset_tb,
102.               clock => clock_tb,
103.               input => output_rom_tb,
104.               output => output_dff_tb
105.           );
106.
107.    sub: Subtractor
108.           generic map (Nbit => 24)
109.           port map (
110.               clock => clock_tb,
111.               reset => reset_tb,
112.               x => output_interpolator_tb,
113.               y => output_dff_tb,
114.               output => difference_tb
115.           );
116.
117.
118.       test_proc: process(reset_tb, clock_tb)
119.           variable count : natural := 0;
120.           begin
121.               if reset_tb = '1' then
122.                   input_tb <= (others => '0');
123.               elsif rising_edge(clock_tb) then
124.                   count := count + 1;
125.           input_tb <= std_logic_vector(unsigned(input_tb) + 1);
126.
127.                   if count = 2**16 then
128.                       end_sim <= '1';
129.                   end if;
130.               end if;
131.           end process;
132.
133.   end Test;
```

The test process changes the input value synchronously with the clock in the range $[0, 2^{16} - 1]$: the input signal is used to drive both the sine interpolator and the ROM, while the output of the two units is fed to a subtractor. Although the ROM is implemented as a synchronous circuit, its output is not provided directly to the subtractor, but by means of a pipelined flip-flop, which works as a delay component to synchronize the output of the interpolator and the one of the ROM. This necessity is due to the sine interpolator providing its output with an embedded flip-flop: using this stratagem, the subtractor effectively processes results related to the same phase input at the same clock cycle.

4.2 Validation

Given the perfect correspondence between the Python simulator and the VHDL code, the validation of the results has been performed directly in Python comparing the values obtained with the interpolation with the ones of the *sin()* function offered by the mathematical library, for all the possible inputs.

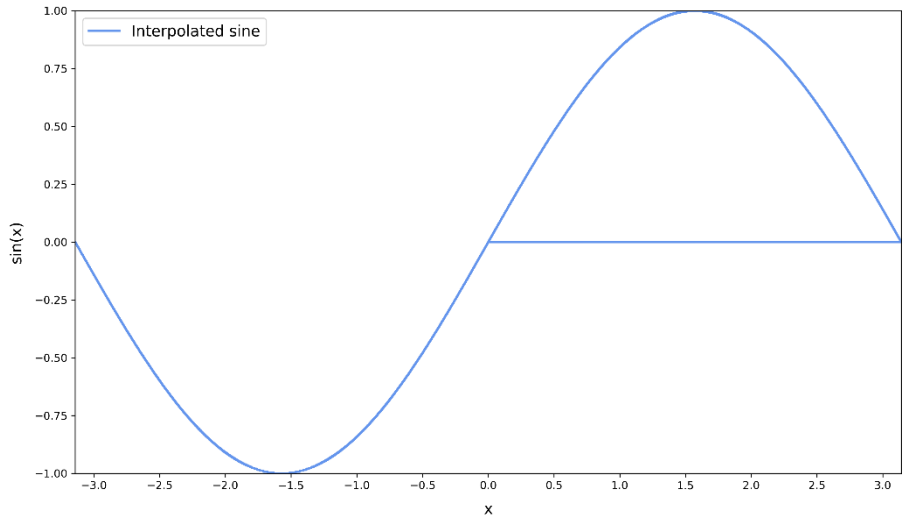

Figure 4 Interpolated sine behaviour obtained with the digital circuit.

The horizontal line in the positive interval is due to the two possible representations of zero in signed fixed-point, namely "100…0" and "000…0". The average error of the approximation is $9.6781 \cdot 10^{-4}$, while the maximum is $1.9594 \cdot 10^{-3}$; the error is uniformly distributed.

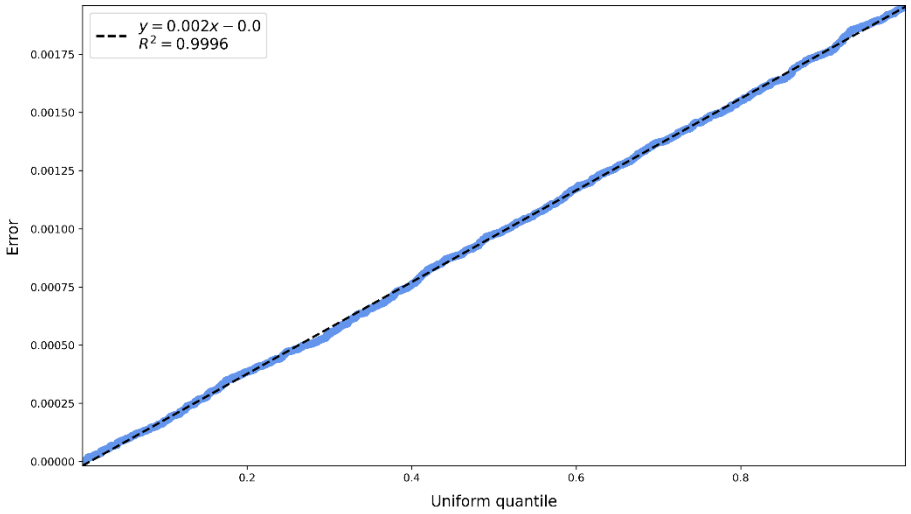

Figure 5 QQ-plot of the error against the quantiles of a uniform distribution.

5 SYNTHESIS AND IMPLEMENTATION

The electronic circuit described in VHDL was synthesized and implemented for the Xilinx FPGA Zync platform by means of the tool VIVADO, specifically targeting the product family Zynq-7000 and the device ZYNQ XC7Z010-1CLG400C-1. The architecture is assumed to be the best one, so no additional optimization steps were requested to the tool in terms of area, speed or power consumption.

5.1 RTL analysis

After importing the VHDL source files, the tool derived the following schematic:

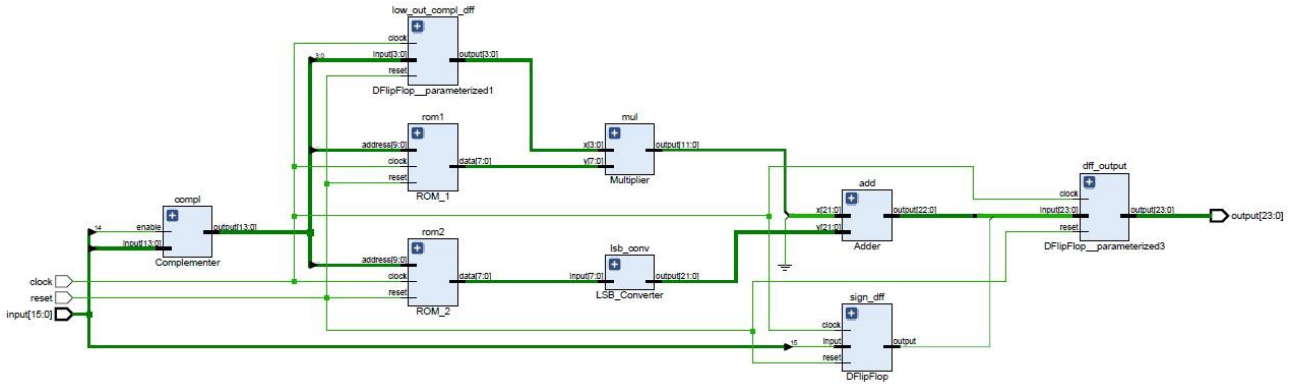

Figure 6 RTL schematic of the sine interpolator generated by VIVADO.

5.2 Synthesis

A first synthesis without constraints was performed, followed by another one with a clock constraint of 125 MHz. The procedure concluded successfully, with only one warning "`[Constraints 18-5210] No constraints selected for write`", which can be safely ignored and indicates that there are no constraints for the design.

The timing report shows that all the timing constraints are met, where the positive slacks mean that no setup or hold violations are detected in this phase; the critical path for t_{SETUP} is the one traversing ROM1, the multiplier and the adder (Figure 8), while the critical path for t_{HOLD} is the one linking the two flip-flops memorizing the sign and the output (Figure 9).

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,953 ns	Worst Hold Slack (WHS):	0,154 ns	Worst Pulse Width Slack (WPWS):	3,500 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	57	Total Number of Endpoints:	57	Total Number of Endpoints:	25

All user specified timing constraints are met.

Figure 7 Timing report obtained in the synthesis phase.

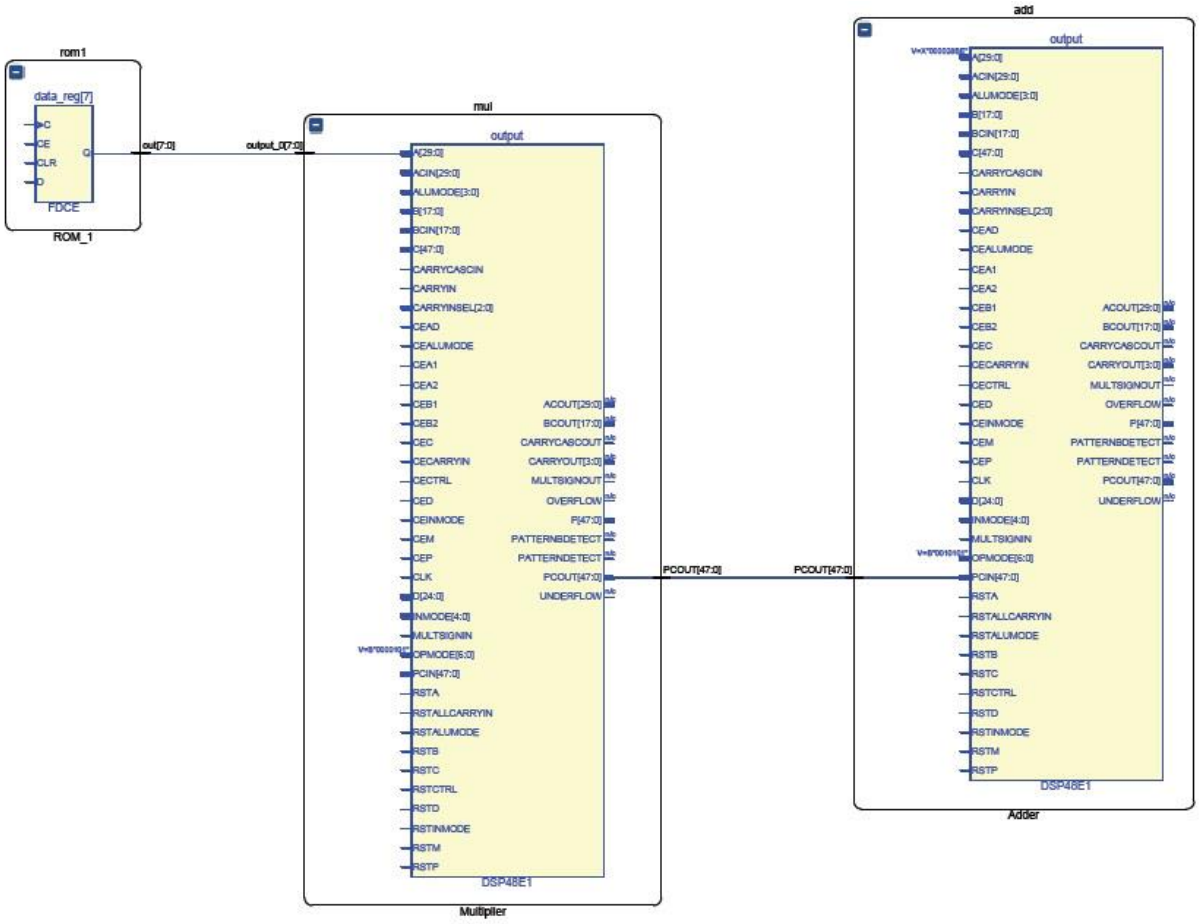

Figure 8 Critical path for t_{SETUP}.

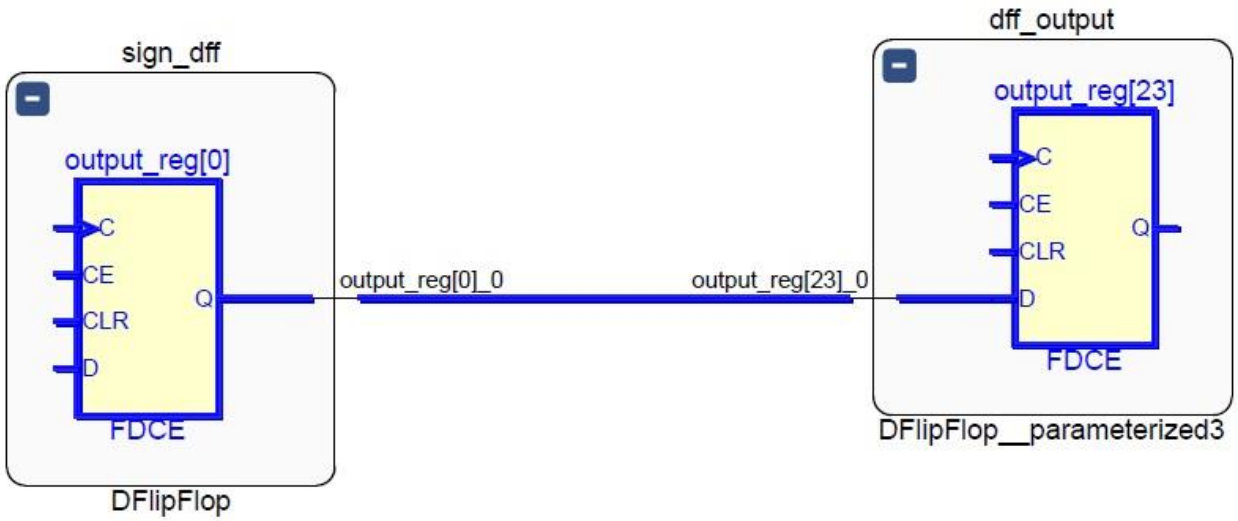

Figure 9 Critical path for t_{HOLD}.

For what concerns the utilization of the FPGA, the report shows that the I/O pin request is the most demanding, taking more than 40% of the available pins, while the other requirements are easily met and consume a small portion of the provided lookup tables, flip-flops and digital signal processors, with a general under-utilization of the overall resources.

Name	Slice LUTs (17600)	Slice Registers (35200)	F7 Muxes (8800)	F8 Muxes (4400)	DSPs (80)	Bonded IOB (100)	BUFGCTRL (32)
SineInterpolator	203	23	54	19	2	42	1
add (Adder)	12	0	0	0	1	0	0
compl (Complementer)	16	0	0	0	0	0	0
dff_output (DFlipFlop__parameterized3)	0	1	0	0	0	0	0
low_out_compl_dff (DFlipFlop__parameterized1)	0	4	0	0	0	0	0
mul (Multiplier)	0	0	0	0	1	0	0
rom1 (ROM_1)	87	8	25	11	0	0	0
rom2 (ROM_2)	88	8	29	8	0	0	0
sign_dff (DFlipFlop)	0	1	0	0	0	0	0

Resource	Utilization	Available	Utilization %
LUT	203	17600	1.15
FF	23	35200	0.07
DSP	2	80	2.50
IO	42	100	42.00

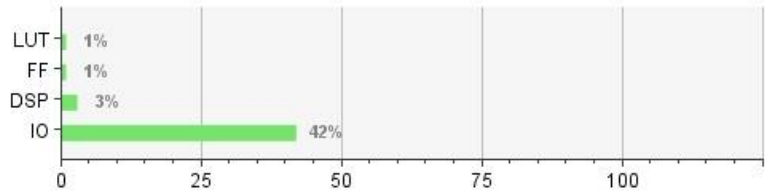

Figure 10 Resource utilization obtained in the synthesis phase.

5.3 Implementation

The implementation phase is fundamental to obtain a more precise evaluation of the behaviour of the circuit in terms of speed and power consumption, at least for the selected board.
The timing report confirms the previous results, i.e. all the timing constraints are met and the slacks are positive. The latter are increased with respect to the synthesis phase:

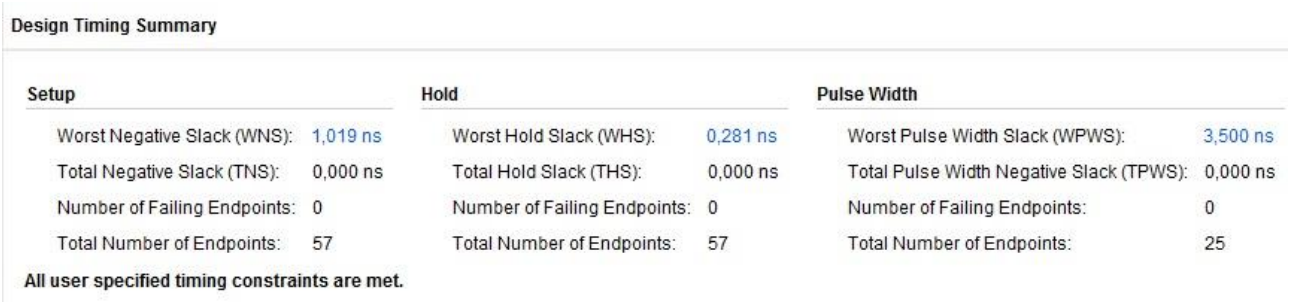

Figure 11 Timing report obtained in the implementation phase.

Therefore, the maximum clock frequency that can be achieved is:

$$T_{CLOCK}^{MIN} = T_{CLOCK} - t_{SLACK} = 8\,ns - 1.019\,ns = 6.981\,ns$$

$$f_{CLOCK}^{MAX} = \frac{1}{T_{CLOCK}^{MIN}} \cong 143.246\,MHz$$

For what concerns the utilization, the results are practically the same, with only one LUT slice saved with respect to the evaluation done in the synthesis (202 instead of 203).
The power report shows a total on-chip power of 0.125 W, with a thermal margin of 58.6°C, where 74% of the power consumption is static and the dynamic one is dominated by the I/O system.

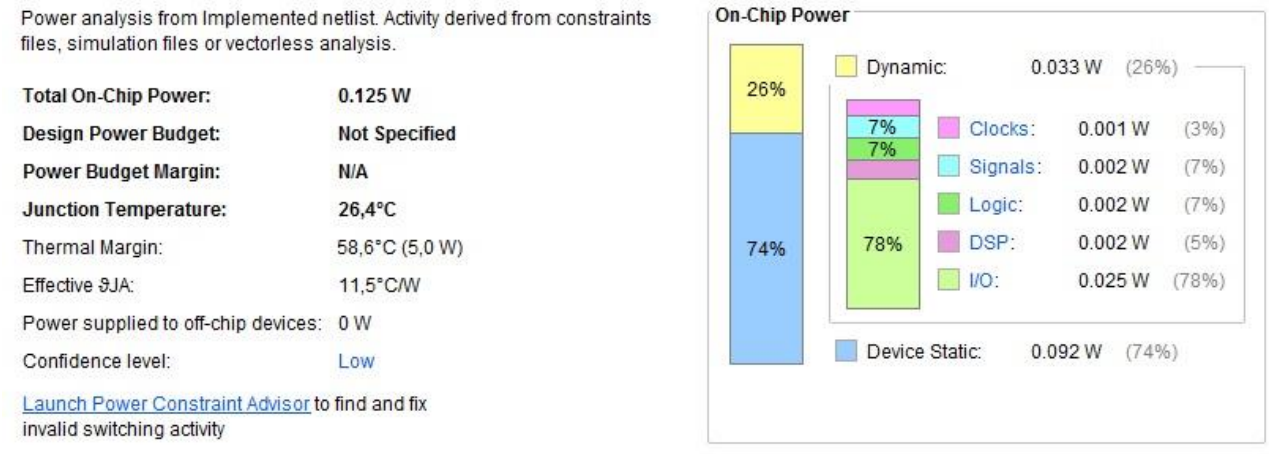

Figure 12 Power report obtained in the implementation phase.

6 CONCLUSION

In this document, a simple and fast electronic circuit for the computation of the sine function using interpolation was presented, together with its VHDL description. The circuit provides approximations with an average and maximum error of $9.6781 \cdot 10^{-4}$ and $1.9594 \cdot 10^{-3}$ respectively, hence providing a good accuracy, although being not suited for high precision applications. Its implementation for the Xilinx FPGA Zync platform on the device ZYNQ XC7Z010-1CLG400C-1 provided an operative maximum clock frequency approximately equal to 143.246 MHz and a total on-chip power of 0.125W, with 74% of the power consumption being static.