CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering

Department of Electric Drives and Traction

Low Abstraction Real-Time FPGA Implementation of Selective Harmonic Elimination Algorithm for Voltage Source Inverters Designed Using State of The Art Free and Open Source Software

Technical report

Petr Zakopal

Prague 2023

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction

This paper presents the design of multiple FPGA units, which are designed to suit near real-time constraints of controlling the electric drives or for Hardware In Loop systems.

The goal of this paper also was to investigate how to design the speed optimized units using open source toolchain. The final designed unit is capable of solving the Selective Harmonic Elimination (SHE) algorithm. Many researches opt for proprietary design software, which very often offers premade Intelectual Property (IP) blocks, which can be used to design the specified circuit. However in this paper the design was created, tested and analyzed solely using the State of The Art Open Source software without any IP catalogs. This platformless solution ensures, that the designed units may possibly be synthetized for various FPGA chips without any major barriers.

The structure of the paper is as follows: Section 3 presents a unit for division of two arbitrary values by utilizing the Newton-Raphson (NR) algorithm. Section 4 presents design of the Coordinate Rotation Digital Computer (SHE) optimized for speed, rather than lesser complexity. Section 5 introduces design which solves two non-linear equations with a Newton-Raphson (NR) algorithm, presenting suitability of FPGA designs for iterative algorithms. Section 6 presents unit for solving the Selective Harmonic Elimination problem using previously developed modules.

2 Note on the circuit Verilog designs

All of the designs presented in this paper are created using pure Verilog code and tested through Free and Open-Source Software (FOSS). The decision to opt for FOSS was deliberate, aiming to prevent any vendor-locking to a specific hardware or predefined IPs. Predefined IPs are often optimized by a specific hardware vendor and intended for use with that vendor's hardware. However, the hardware may not always be available or suitable for the application.

Once the design and algorithm are thoroughly understood, they can be implemented without any specific platform in mind. Later, when selecting the device vendor, the design can be modified to suit the specific hardware requirements. That is why Verilog, with Cocotb [1] (Test Bench creation tool) and Verilator [2] (simulator) were used for designing the circuits presented in this paper.

3 Calculating the division of fixed point numbers

Typically, when employing numerical methods to solve transcendental equations, the calculation of the division of two input numbers becomes necessary. This requirement persists even when applying the Newton-Raphson (NR) method to solve a set of two equations, because computing the reciprocal value of the Jacobian determinant is necessary.

There are some IP blocks available, which are capable of calculating the division of two numbers, but the blocks are usually either vendor specific intellectual property (IP) [3] or feature low performance [4].

The drawback of vendor-specific IPs lies in their limited compatibility, often preventing their use with FPGA chips from different vendors. On the other hand the vendor specific IPs are usually optimized and able to use the specific type of resources available at the vendor's chip which resolve in better performance.

To preserve the compatibility of the design with chips from multiple vendors, the custom solution for division design based on the very well known Newton Raphson (NR) algorithm was developed. [4]

3.1 Newton Rapshon algorithm for calculating the division

General Newton Raphson (NR) algorithm is a well known approach to numerically solve equations. It is the reason why it is utilized in many algorithms. However, the negative aspect of NR is that it's convergency strongly depends on initial values of variables. When the initial values are chosen poorly, the performed number of iterations before the convergency is reached can be high.

To reach the fastest convergency possible (determined in number of iterations) apart from the scaling the dominator into the interval [0.5,1] the initial value calculation formula should be utilized. [4]

The Equation 3 - 1 for calculating the initial value is applied after the scaling of denominator is performed. The algorithm developed for the appropriate scaling is explained in the *Calculating number of bits to shift the denominator*.

$$x_0 = \frac{48}{17} - \frac{32}{17}D, \tag{3 - 1}$$

where the x_0 is the initial value for the NR algorithm, D is the denominator value for calculating the expression N/D, where N is the numerator.

Because in the module design implemented via Verilog the fixed point number format *Q32.15* is used, the fractional numbers from Equation 3 - 1 are rounded to
2.8229 (32'sb00000000000000010_110100101011000 in binary)
and 1.8819 (32'sb00000000000000001_111000011100101 in binary) respectively.

After the initial value x_0 is calculated, the NR algorithm is performed. The idea of using NR algorithm to calculate the division of N/D is to trade the division for a multiplication which can be synthetized in the FPGA fabric. When employing the NR algorithm for finding the values of N/D the function with root is $1/D$ is essential. After the root of the function is found, it is then multiplied by the numerator value, and te solution N/D is obtained. There may be many functions, which root is the searched value $1/D$ but the most trivial is Equation 3 - 2.

$$\mathrm{F}(x_i) = \frac{1}{x_i} - D. \tag{3 - 2}$$

For the derivative at the point of x_i then applies Equation 3 - 3.

$$\frac{\mathrm{d}F(x_i)}{\mathrm{d}x} = F'(x_i) = \frac{F(x_{i+1}) - F(x_i)}{x_{i+1} - x_i}. \tag{3 - 3}$$

Because finding root of the Equation 3 - 2, the value of $F(x_{i+1})$ is set to be zero. After separating the x_{i+1} value of the Equation 3 - 3 and derivating the function $F(x_i)$ the obtained algorithm for a value x_{i+1} is obtained from eq. 3 - 4.

$$x_{i+1} = -\frac{F(x_i)}{F'(x_i)} + x_i = -\frac{F(x_i)}{-\frac{1}{x_i^2}} + x_i = (\frac{1}{x_i} - D)x_i^2 + x_i = x_i - Dx_i^2 + x_i = 2x_i - Dx_i^2. \tag{3 - 4}$$

Usually, the iterative algorithm is stopped, when the value $F(x_{i+1}) - F(x_i)$ (called the defect) reaches certain value set by the stop condition. However, in this algorithm, the stop condition is not yet implemented. Based on the observation carried on the NR algorithm the obtained result is sufficient after 5 iterations.

The mathematically expressed algorithm is then transformed into programmable algorithm suitable for FPGA implementation. The top module design for this algorithm are presented in the section *Top module design*, the control and data unit for calculating the value x_{i+1} is presented in the *Allocation and Timing*

3.2 IP Block Design

The design of this unit consists of 4 main modules:

- the **data unit module**, used for manipulating data and making calculation operations,
- the **control unit module**, used for controlling the **data unit module** and **scaling unit module**,
- **scaling unit module**, used for calculating the number of bits needed for shifting the denominator value to the interval [0.5,1].

3.2.1 Top module design

The top module wraps all of the presented modules (**data unit module**, **control unit module**, **scaling unit module**). The basic structure of connected modules of this top design is depicted in the Figure 3 - 1. Thanks to this wrapper it is possible to test the created modules with Verilog Testbench, Verilator [2] or Cocotb [1].

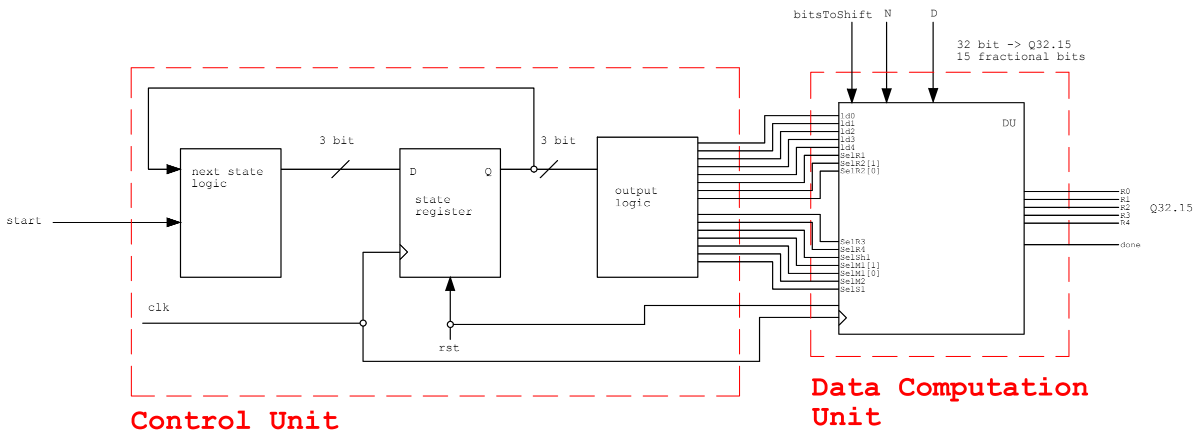


Figure 3 - 1 Top module design for the division unit module block

3.2.2 Allocation and Timing

The diagram of the data flow and timing of the algorithm is displayed in the Figure 3 - 2.

The complete algorithm comprises nine steps. The initial four steps are used for calculating the initial value of x_0 as presented in the Equation 3 - 1. The steps *S4* to *S8* are for calculating the next search value of x_{i+1}, thus the root of the Equation 3 - 2 which in fact is the searched value of $1/D$. The following iteration begins at the step labeled as *S5*. The iterative process continues until a predefined stop condition is satisfied, such as reaching a specified number of iterations.

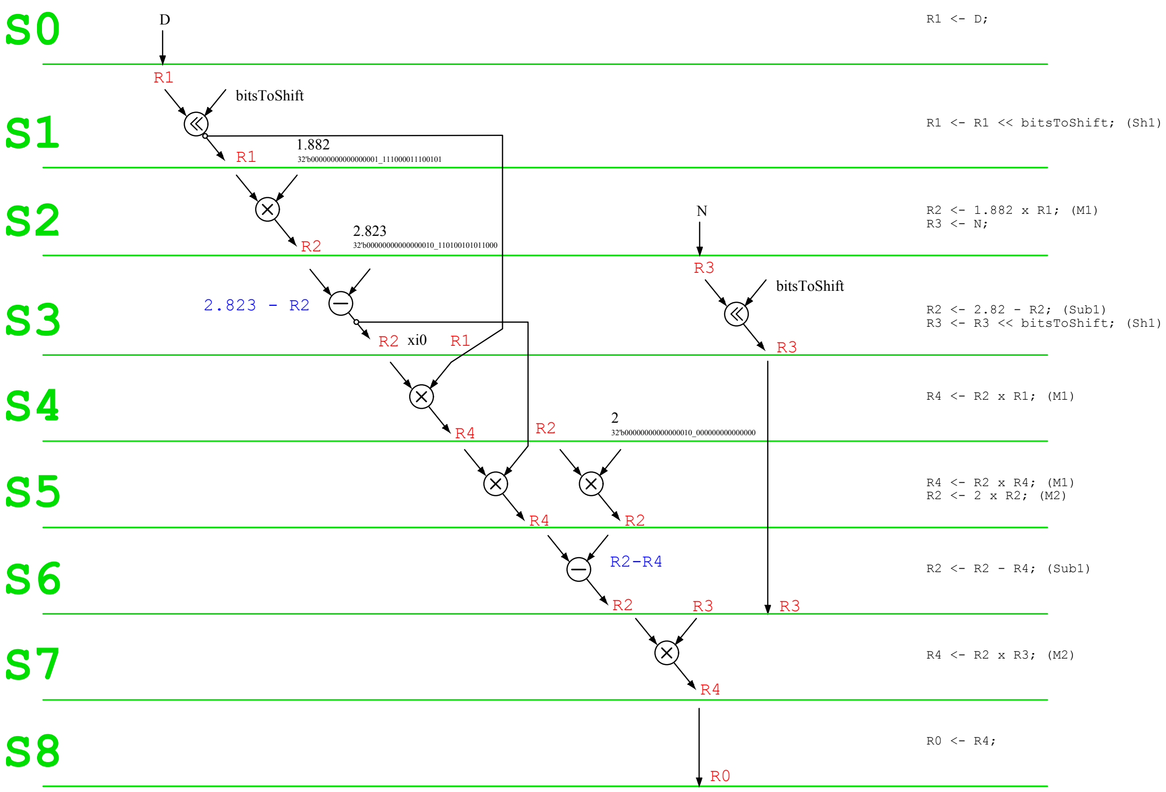


Figure 3 - 2 Allocation and timing diagram for the Data Path part of the division module.

3.2.3 Data Path Module

The structure of the Data Path Module is depicted in the Figure 3 - 3. The module was specifically designed to serve the needs of the division algorithm. It comprises five registers labeled $R0$ through $R4$, two multipliers $M1$, $M2$ and one bit shifter.

The module is controlled by the control unit using the control signal labeled as CS. The encoding table with the labels corresponding to the Data Path Unit module is presented in the section *Control Unit*.

The result of each iteration from the division algorithm is passed to a register $R0$.

The Data Path Module unit also covers the possibility of using negative denominator and numerator. Because the values are stored in a custom *Q32.15* fixed point format (whole number comprises of 32 bits, 15 bits fractional part, 17 bits integer part), the algorithm checks if the D or N values are higher than $0h8000$ and determine it's actual sign and the sets sign of the result. If the analyzed number is determined negative, it is transformed to value positive and then used in the presented division algorithm. This transformation is needed because of the algorithm calculating the bits to shift the denominator in the interval.

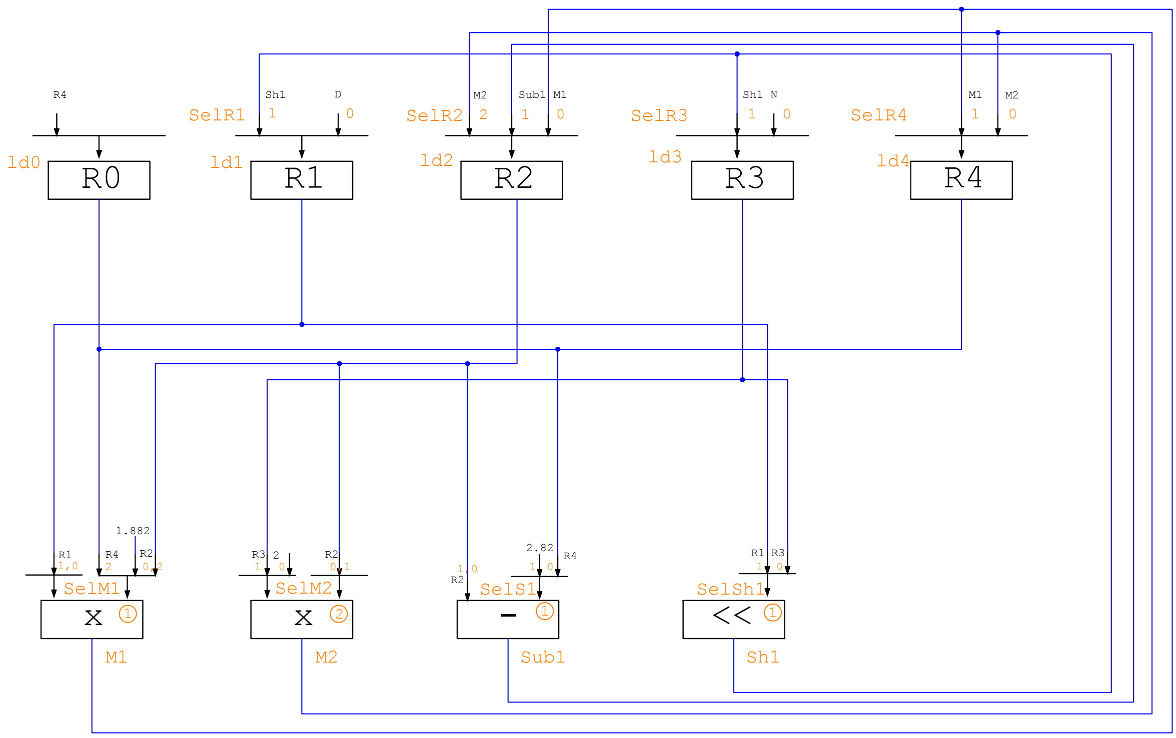


Figure 3 - 3 Register Transfer Level (RTL) scheme of the Division module Data Path.

3.2.4 Control Unit

Signals from Control Unit to Data Path Module are encoded in the CS signal. Table 3 - 1 displays the CS signal along with the corresponding instructions for steps $S0$–$S8$ of the FSM. To enhance code clarity the signal is defined in the Control Unit in the hexadecimal format.

The number of the iteration of the Finite State Machine (FSM) is also set in the Control Unit. This iteration number is subsequently used in the module to check for the stop condition of the calculation loop.

As stated in the *Allocation and Timing* section, after the step *S8*, the FSM restarts at the state *S4* with new x_i values as inputs. This state change is not depicted in the Table 3 - 1 for CS signal.

Table 3 - 1 Control signal encoding table for instructions to be processed by the Division Module.

State	RTL Code	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	CS
		ld0	ld1	ld2	ld3	ld4	SelR1	SelR2[1]	SelR2[0]	SelR3	SelR4	SelSh1	SelM1[1]	SelM1[0]	SelM2	SelS1	
S0	R1 ← D;	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	2000h
S1	R1 ← R1 « 32; (Sh1)	0	1	0	0	0	1	0	0	0	0	1	0	0	0	0	15’h2210
S2	R2 ← 1.882 x R1; (M1) R3 ← N;	0	0	1	1	0	0	0	0	0	0	0	0	1	0	0	15’h1804
S3	R2 ← 2.82 - R2; (Sub1) R3 ← R3 « 32; (Sh1)	0	0	1	1	0	0	0	1	1	0	0	0	0	0	0	15’h18C0
S4	R4 ← R2 x R1; (M1)	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	420h
S5	R4 ← R2 x R4; (M1) R2 ← 2 x R2; (M2)	0	0	1	0	1	0	1	0	0	1	0	1	0	0	0	15’h1528
S6	R2 ← R2 - R4; (S1)	0	0	1	0	0	0	0	1	0	0	0	0	0	0	1	15’h1081
S7	R4 ← R2 x R3; (M2)	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	15’h402
S8	R0 ← R4;	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4000h

3.3 Calculating number of bits to shift the denominator

As presented in the section *Newton Rapshon algorithm for calculating the division* the denominator must be appropriately scaled for the division algorithm to work. This section presents algorithm for scaling the denominator specified in the fixed point number format *Q32.15*. After the scaling value is successfully determined, the numerator is scaled accordingly.

The presented algorithm shifts the value of denominator at every positive edge of the clock signal and saves the shifted value in the `compare` register. Then the combinational circuit is utilized to compare the shifted value in `compare` register with the number `1` specified in *Q32.15* format. If the compared value is the same or lower than `1` the shifting algorithm is done and the value `scaleToShift` is successfully found. If not, the inner value of shifting bits is incremented and the algorithm proceeds to the next iteration.

The presented algorithm is realized in the *denominatorSizeScaleUnit* module and it's pseudocode is depicted in the code 3 - 1.

```
at every negative edge of clock or positive edge of reset
    if(rst)
        scaleToShift = 0;
        scaleToShiftInternal = 1;
        started = 0;
    end if
    else if (start)
        started = 1;
    end else if

at every positive edge of clock
    if (compare <= 32'b00000000000000001_000000000000000)
        done = 1;
        started = 0;
        scaleToShift = scaleToShiftInternal;
    end if
    else
        done = 0;
        scaleToShiftInternal = scaleToShiftInternal + 1;
    end if

at every positive edge of clock
    if(start)
    compare <= DInternal >> scaleToShiftInternal;
    end if
```

Code 3 - 1 Pseudocode for the denominatorSizeScaleUnit module algorithm.

3.4 Simulation results

The simulation via Verilog testbench was made to determine the correctness of presented division module. The Icarus Verilog simulator was used to simulate the module and GTKWave was used to display the VCD simulation output file.

The simulation output confirms that the module operates correctly for positive and negative numbers in the fixed-point format *Q32.15*. The algorithm used in this module can compute the correct result in significantly fewer clock cycles compared to the full division algorithm utilized in the division module within the package [4]. As a result, the module can be freely used as a submodule in more complex modules.

The rendered VCD simulation output waveforms are depicted on the following Figures. The simulations were conducted for arbitrarily selected values of numerator N and denominator D, with clock frequency set to 250 MHz. Pseudocode Verilog snippet for the test bench is provided in the Listing 3 - 2. In the test bench, one unit of time corresponds to 1 ns. (based on the set timescale settings) The division unit algorithm starts at the next positive edge of clock signal after successful determination of the value *bitsToShift*.

```
 timescale 1ns/1ns
 #10; // wait for 10 units of time
 #0 rstScale = 1; startScale = 0; // reset unit for determining the number of bits to shift in the denominator and do not start the unit yet
 N = 32'b00000000100110000_000010000000000; D=32'b11111111111111111_110000000000000; // set the numerator to N = 304.03125, denominator to D = -0.25
 #10 rstScale = 0; // wait for 10 units of time and stop the reset of scaling unit
 #10 startScale = 1; // start the algorithm for scaling unit
 #20 rst = 1; start = 0; // reset the division unit
 #30 rst = 0; // stop reseting of the division unit
 #20 start = 1; // start the division unit
 #20 start = 0;
 #1000; // wait 1000 units of time
 $finish; // finish the simulation
```

Code 3 - 2 Pseudocode snippet for the Verilog simulation test bench.

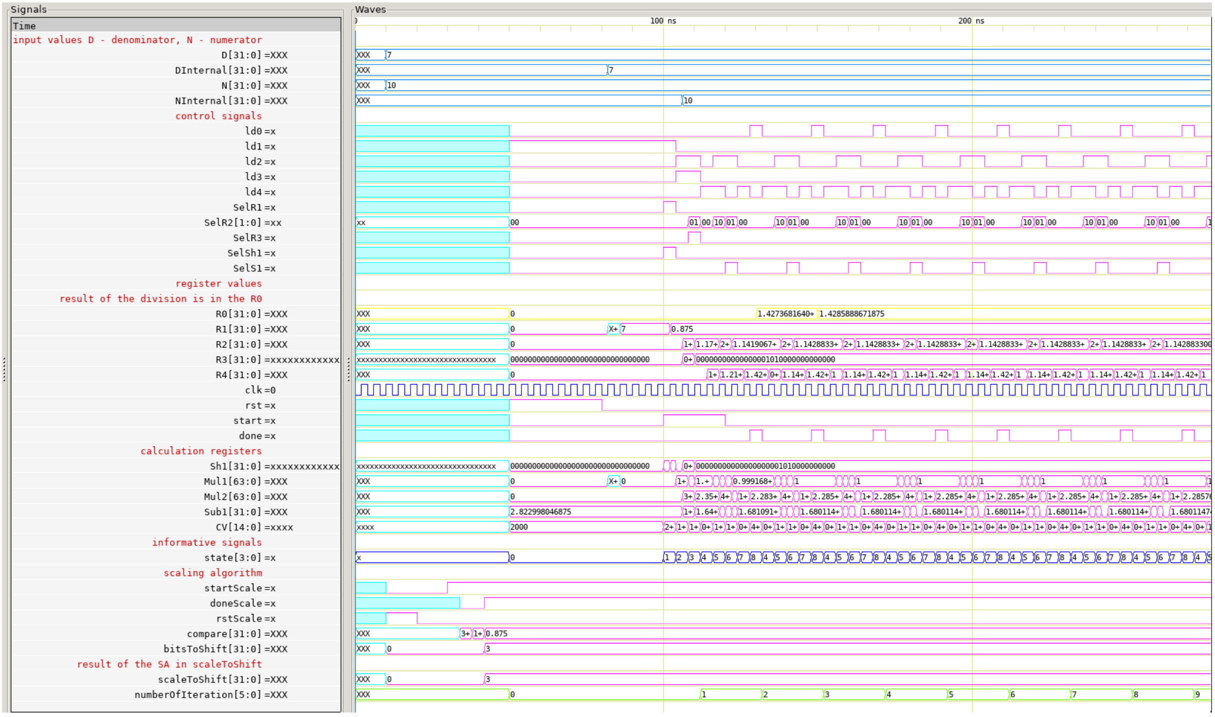


Figure 3 - 4 Selected signals from simulation of division N/D = 10 / 7. The correct result in R0 is obtained after two iterations (register numberOfIterations).

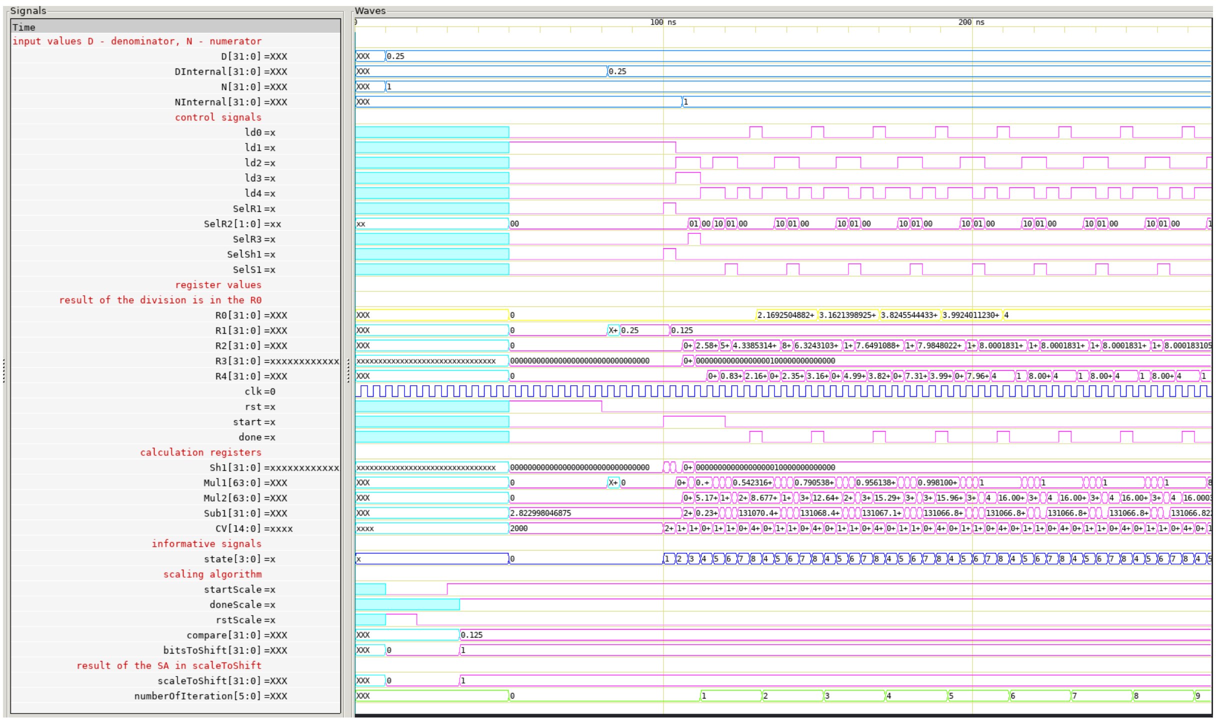


Figure 3 - 5 Selected signals from simulation of division N/D = 1 / 0.25. The correct result in R0 is obtained after five iterations (register numberOfIterations).

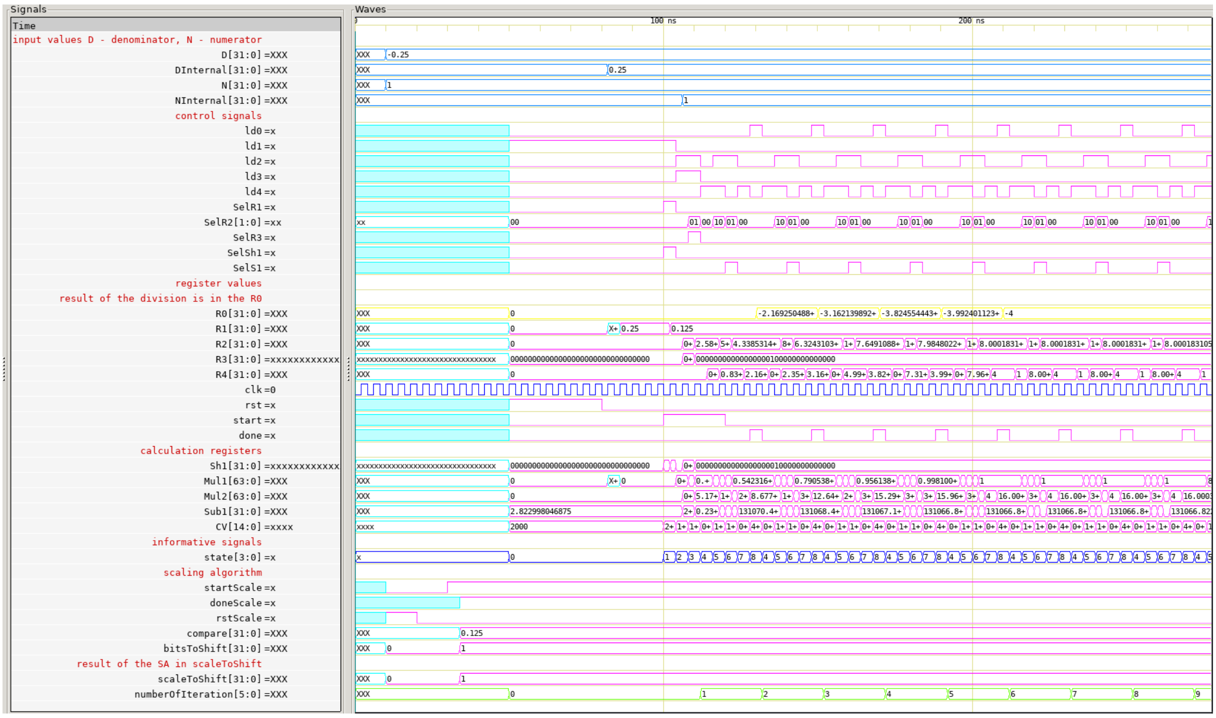


Figure 3 - 6 Selected signals from simulation of division N/D = 1 / (-0.25). The correct result in R0 is obtained after five iterations (register numberOfIterations).

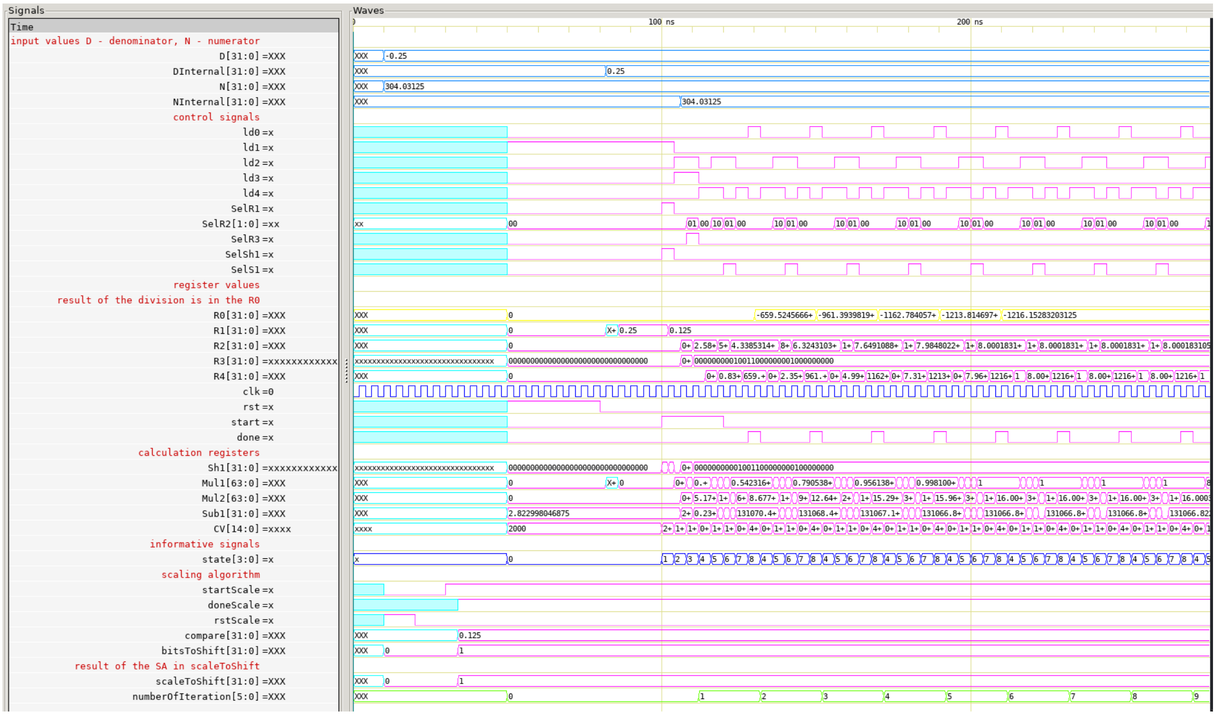


Figure 3 - 7 Selected signals from simulation of division N/D = 304.03215 / (-0.25). The correct result in R0 is obtained after five iterations (register numberOfIterations).

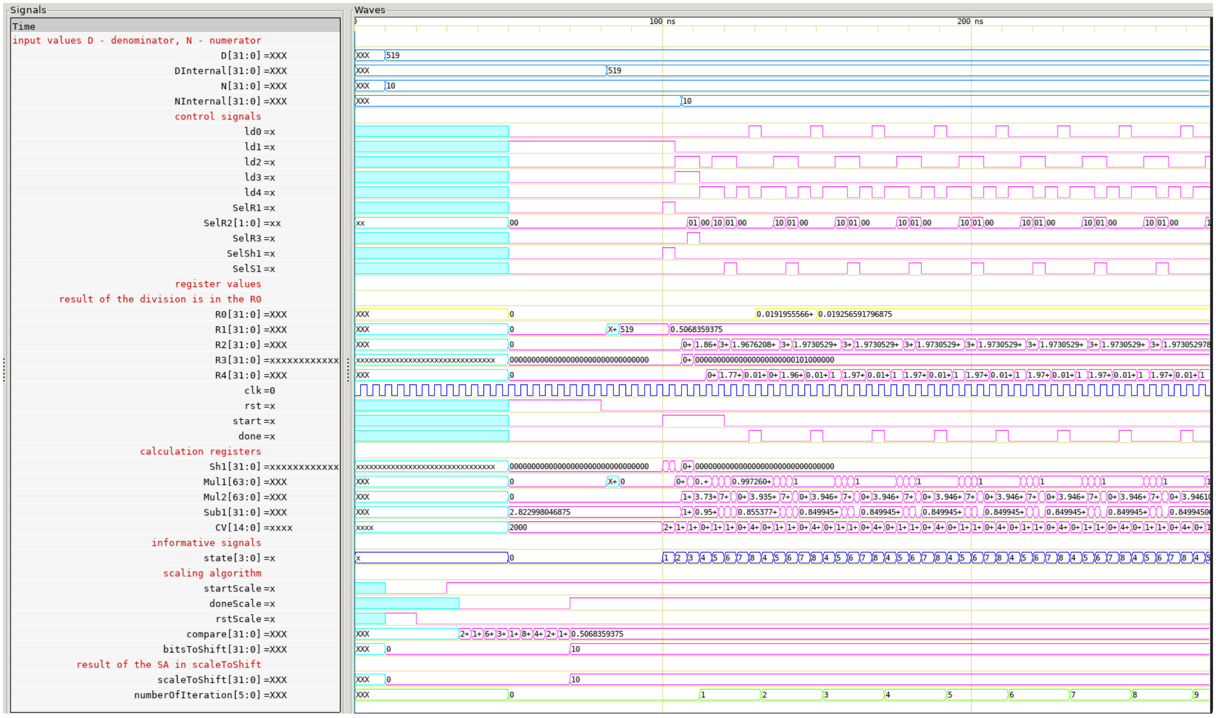


Figure 3 - 8 Selected signals from simulation of division N/D = 10 / 519. The correct result in R0 is obtained after two iterations (register numberOfIterations).

4 Using CORDIC to calculate trigonometric functions

There are numerous methods calculating trigonometric functions. To enhance flexibility of the design, the Coordinate Rotation Digital Computer (CORDIC) was selected over the Look-Up Table (LUT) implementation.

While the LUT method may be fast, its accuracy depends on the size of the table. In contrast, when using the CORDIC the precision depends on number of performed iterations of the algorithm. The modified algorithm is versatile and may be used to calculate non-trivial functions, including hyperbolic functions, square roots, multiplications, divisions, exponentials and logarithms. [5]

In this work only the calculation of $sine$ and $cosine$ functions is performed.

4.1 Theory

The theory of the first CORDIC was introduced by Volder in [6]. This algorithm computes a coordinate conversion between rectangular (x, y) and polar (R, θ) coordinates. The algorithm was then extended by Walther in [7] to include circular, linear and hyperbolic transforms. In this paper, only circular transforms are employed to calculate $sine$ and $cosine$ functions. The presentation will focus on the fundamental aspects of the algorithm.

The rotation of a vector in the rectangular coordinate system (x, y) may be described by matrix-vector multiplication depicted in the Equation 4 - 1.

$$\begin{pmatrix} x_\mathrm{R} \\ y_\mathrm{R} \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} x_\mathrm{in} \\ y_\mathrm{in} \end{pmatrix}, \tag{4 - 1}$$

where x_R and y_R are coordinates of a rotated vector, θ is the angle for which the vector with coordinates x_in and y_in is rotated.

Then when simplifying the Equation 4 - 1

$$\begin{pmatrix} x_\mathrm{R} \\ y_\mathrm{R} \end{pmatrix} = \cos(\theta) \begin{pmatrix} 1 & -\tan(\theta) \\ \tan(\theta) & 1 \end{pmatrix} \begin{pmatrix} x_\mathrm{in} \\ y_\mathrm{in} \end{pmatrix} \tag{4 - 2}$$

it can be seen, that only multiplication by scaling factor of precalculated values of $\cos(\theta)$, multiplication by $\tan(\theta)$, subtraction and addiction operations are needed to perform the rotation. However, the multiplication by $\tan(\theta)$ can be replaced. The replacement may be done for angles θ for which the Equation 4 - 3 is true. When implementing the algorithm to the FPGA the multiplication may be swapped for signed right bit shift, which is faster operation than multiplication.

$$\tan(\theta) = 2^{-1}. \tag{4 - 3}$$

When the initial values $x_\mathrm{in} = 1$ and $y_\mathrm{in} = 0$ are used, the result for $sine$ and $cosine$ may be easily obtained from x_R and y_R as expressed in the Equation 4 - 4.

$$\begin{aligned} x_\mathrm{R} &= x_\mathrm{in}\cos(\theta) - y_\mathrm{in}\sin(\theta) = \cos(\theta), \\ y_\mathrm{R} &= x_\mathrm{in}\sin(\theta) + y_\mathrm{in}\cos(\theta) = \sin(\theta). \end{aligned} \tag{4 - 4}$$

The algorithm can be further simplified by assuming that it is designed to undergo more than 6 iterations and thus the scaling constant, represented by multipliying $cosine$ of different θ values, converges to the value 0.60725. If this condition is true, there is no necessity to precalculate all the scaling values and only the convergenent value may be used for the multiplication. In this paper the precalculated scaling values are passed from the custom LUT module to the main algorithm.

As evident from the *Example of calculation* section or the algorithm theory itself, it is essential to estabilish whether the angle for which the vector is rotated in the next iteration should be in a positive direction (counter-clockwise) or negative direction (clockwise). To address this, the set of the equations is expanded, and new variable z_i is introduced. The complete set of equations utilized in the implementation is as follows:

$$\begin{aligned} x[i+1] &= x[i] - \sigma_i 2^{-i} y[i], \\ y[i+1] &= y[i] + \sigma_i 2^{-i} x[i], \\ z\,[i+1] &= z\,[i] - \sigma_i \operatorname{atan}\left(2^{-i}\right). \end{aligned} \tag{4 - 5}$$

The σ_{i+1} is determined based on the sign of the z_{i+1} variable

$$\sigma_{i+1} = \left\{ \begin{array}{l} -1, \text{ if } z_{i+1} < 0 \\ 1, \text{ if } z_{i+1} > 0 \\ 0, \text{ if } z_{i+1} = 0 \end{array} \right\} \tag{4 - 6}$$

The algorithm, as presented, accurately computes values for $sine$ and $cosine$ functions only in the first and fourth quadrants ($-\pi/2$ to $\pi/2$ counter-clockwise). To expand its applicability across the entire 2π range, specific actions must be taken before the actual looped aglorithm.

The algorithm must determine the quadrant, where the desired angle θ for which the $sine$ and $cosine$ functions are to be calculated is. This determination is made through `if` statements during the initialization of the algorithm values and at the final value calculation. If the reference angle θ falls outside the first or fourth quadrant, then the angle is rotated from its original quadrant to either the first or fourth quadrant. Depending on the quadrant, to which the angle is rotated, the σ_i value is set accordingly. The corresponding `if` statements during the algorithm initialization are provided in Pseudocode 4 - 1. Similar statements used at the final values calculation are presented in Pseudocode 4 - 2.

The pseudocodes use *initialZValue* as a reference angle θ, for which to calculate the $sine$ and $cosine$ function values, *zValue* as a temporary value for calculating the iterations for z_i variables, *sigmaValue* for temporary value, which holds the current iteration value of σ_i, the *resultCos* and *resultSin* variables are used for storing the temporary and final values of the $\cos(\theta)$ and $\sin(\theta)$ values respectively.

```
if (initialZValueCordic > 1.5707) and (initialZValueCordic < 3.141592):
    zValue = initialZValueCordic - 3.141592
    sigmaValue = -1
    print("value in second q")
    print("zValue:", zValue)
elif (initialZValueCordic > 3.141592) and (initialZValueCordic < 4.7123):
    zValue = initialZValueCordic - 3.141592
    sigmaValue = 1
    print("value in third q")
    print("zValue:", zValue)
elif (initialZValueCordic < 0) and (initialZValueCordic > - 1.5707):
    sigmaValue = -1
    zValue = initialZValueCordic
```

```
    print("value in fourth q")
    print("zValue:", zValue)
elif (initialZValueCordic < -1.5707) and (initialZValueCordic > - 3.141592):
    sigmaValue = 1
    zValue = initialZValueCordic + 3.141592
    print("value in third q")
    print("zValue:", zValue)
elif (initialZValueCordic < - 3.141592) and (initialZValueCordic > - 4.7129):
    sigmaValue = - 1
    zValue = initialZValueCordic + 3.141592
    print("value in second q")
    print("zValue:", zValue)
elif (initialZValueCordic < - 4.7129) and (initialZValueCordic > - 6.28318):
    sigmaValue = initialSigmaValueCordic
    zValue = initialZValueCordic + 2*3.141592
    print("value in first q")
    print("zValue:", zValue)
elif (initialZValueCordic > 4.7123) and (initialZValueCordic < 6.28318):
    sigmaValue = -1
    zValue = initialZValueCordic - 2*3.141592
    print("value in fourth q")
    print("zValue:", zValue)
else:
    zValue = initialZValueCordic  # For angle
    sigmaValue = initialSigmaValueCordic  # For +- next angle
    print("value in first")
    print("zValue:", zValue)
```

Code 4 - 1 Pseudocode for if statements used at the value initialization of the CORDIC algorithm.

```
if (initialZValueCordic > 1.5707) and (initialZValueCordic < 3.141592):
    resultCos = - resultCos
    resultSin = - resultSin
elif (initialZValueCordic > 3.141592) and (initialZValueCordic < 4.7123):
    resultCos = - resultCos
    resultSin = - resultSin
elif (initialZValueCordic < 0) and (initialZValueCordic > - 1.5707):
    resultCos = resultCos
    resultSin = resultSin
elif (initialZValueCordic < -1.5707) and (initialZValueCordic > - 3.141592):
    resultCos = -resultCos
    resultSin = -resultSin
elif (initialZValueCordic < - 3.141592) and (initialZValueCordic > - 4.7129):
    resultCos = - resultCos
```

```
    resultSin = - resultSin
elif (initialZValueCordic < - 4.7129) and (initialZValueCordic > -
    2*3.141592):
    resultCos = resultCos
    resultSin = resultSin
```

Code 4 - 2 Pseudocode for if statements used at the final sinus and cosinus value calculation.

4.1.1 Example of calculation

The CORDIC algorithm's general approach can be illustrated by calculating the $sine$ and $cosine$ values for the reference angle $\theta = 57,535\ ^\circ$ (1.0041 rad). Initially, the angle is decompositioned into its base angles, satisfying the Equation 4 - 3. In this example the decomposition is $57,535 = 45{+}25,565{-}14,03$.

The index i of the variables x_i and y_i in the following equations means the number of iteration of the algorithm.

$$\text{0. iteration} \quad \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} = \cos(45\ ^\circ) \begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x_{\text{in}} \\ y_{\text{in}} \end{pmatrix}, \tag{4 - 7}$$

$$\text{1. iteration} \quad \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \cos(26,565\ ^\circ) \begin{pmatrix} 1 & -2^{-1} \\ 2^{-1} & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}, \tag{4 - 8}$$

$$\text{2. iteration} \quad \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \cos(-14,03\ ^\circ) \begin{pmatrix} 1 & -2^{-2} \\ 2^{-2} & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}. \tag{4 - 9}$$

Then values x_2 and y_2 may be obtained.

$$\begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \cos(45\ ^\circ)\cos(25,565\ ^\circ)\cos(-14,03\ ^\circ) \begin{pmatrix} 1 & -2^{-2} \\ 2^{-2} & 1 \end{pmatrix} \begin{pmatrix} 1 & -2^{-1} \\ 2^{-1} & 1 \end{pmatrix} \begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x_{\text{in}} \\ y_{\text{in}} \end{pmatrix}. \tag{4 - 10}$$

The values x_2 and y_2 in the Equation 4 - 10 correspond to $\cos(57,535\ ^\circ)$ and $\sin(57,535\ ^\circ)$ respectively.

4.2 Python Implementation

For simplicity, the CORDIC algorithm was prototyped in Python. This proved highly beneficial, as the debugging of the Python code is much more straightforward compared to debugging the Verilog design without prepared and debugged algoritm in a higher level language.

The Python code was used to precalculate the LUT for scaling factor and *arcus tangens* values for z_i calculations.

For clarity, the Python implementation is provided in Code 4 - 3. The presented Code also calculates the error between the CORDIC-calculated value and the Python math library functions.

```
import math


# Defining starting values and empty arrays
totalNumberOfIterations = 12 # 12 - best tradeof between value and
    iterations
```

```
atanValues = []
scalingValues = [1]
initialXValueCordic = 1
initialYValueCordic = 0
# initialZValueCordic = 1.248  # angle for which to calculate cordic
# initialZValueCordic = - 1.248  # angle for which to calculate cordic
# initialZValueCordic = - 6.7194  # angle for which to calculate cordic
# initialZValueCordic = 6.7194  # angle for which to calculate cordic
# initialZValueCordic = 5.8  # angle for which to calculate cordic
# initialZValueCordic = 10.7194824  # angle for which to calculate cordic
# initialZValueCordic = - 10.7194824  # angle for which to calculate cordic
# initialZValueCordic = 5.8  # angle for which to calculate cordic
# initialZValueCordic = - 5  # angle for which to calculate cordic
# initialZValueCordic = - 5  # angle for which to calculate cordic
# initialZValueCordic = - 20.3948  # angle for which to calculate cordic
# initialZValueCordic = - 1.8  # angle for which to calculate cordic
# initialZValueCordic = 1.6  # angle for which to calculate cordic
# initialZValueCordic = 1.8  # angle for which to calculate cordic
# initialZValueCordic = 3.5  # angle for which to calculate cordic
initialZValueCordic = - 3.5  # angle for which to calculate cordic
initialSigmaValueCordic = 1

for x in range(totalNumberOfIterations):
    # Generating arcus tanges values of precalculated angles based on number of iterations
    atanValues.append(math.atan(1*2**(-x)))
    # Generating precalculated scaling values based on a number of iterations
    scalingValues.append(scalingValues[x]*math.cos(atanValues[x]))

print("atanValues: ", atanValues)
print("scalingValues: ", scalingValues)

print("*-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-*")
print("\n")
print("initialZValue original: ", initialZValueCordic)

# Moving angle to interval [0,2Pi]
if initialZValueCordic > 0:
    while initialZValueCordic > (2*3.141592):
        initialZValueCordic = initialZValueCordic - 2*3.141592
else:
    while initialZValueCordic < (-2*3.141592):
        initialZValueCordic = initialZValueCordic + 2*3.141592


print("initialZValue after moving to [0,2Pi] interval: ",
    initialZValueCordic)
```

```python
print("\n")
print("*-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-*")

# Checking the initial value and moving it in the interval
if (initialZValueCordic > 1.5707) and (initialZValueCordic < 3.141592):
    zValue = initialZValueCordic - 3.141592
    sigmaValue = -1
    print("value in second q")
    print("zValue:", zValue)
elif (initialZValueCordic > 3.141592) and (initialZValueCordic < 4.7123):
    zValue = initialZValueCordic - 3.141592
    sigmaValue = 1
    print("value in third q")
    print("zValue:", zValue)
elif (initialZValueCordic < 0) and (initialZValueCordic > - 1.5707):
    sigmaValue = -1
    zValue = initialZValueCordic
    print("value in fourth q")
    print("zValue:", zValue)
elif (initialZValueCordic < -1.5707) and (initialZValueCordic > -3.141592):
    sigmaValue = 1
    zValue = initialZValueCordic + 3.141592
    print("value in third q")
    print("zValue:", zValue)
elif (initialZValueCordic < - 3.141592) and (initialZValueCordic > -4.7129):
    sigmaValue = - 1
    zValue = initialZValueCordic + 3.141592
    print("value in second q")
    print("zValue:", zValue)
elif (initialZValueCordic < - 4.7129) and (initialZValueCordic > -6.28318):
    sigmaValue = initialSigmaValueCordic
    zValue = initialZValueCordic + 2*3.141592
    print("value in first q")
    print("zValue:", zValue)
elif (initialZValueCordic > 4.7123) and (initialZValueCordic < 6.28318):
    sigmaValue = -1
    zValue = initialZValueCordic - 2*3.141592
    print("value in fourth q")
    print("zValue:", zValue)
else:
    zValue = initialZValueCordic  # For angle
    sigmaValue = initialSigmaValueCordic  # For +- next angle
    print("value in first")
    print("zValue:", zValue)

```

```
# Passing starting values to the calculation values
xValue = initialXValueCordic  # For cos
yValue = initialYValueCordic  # For sin


# CORDIC ALGORITHM
for x in range(totalNumberOfIterations):

    # Calculating next values of the current iteration x
    xNextValue = xValue - (sigmaValue*yValue)*2**(-x)
    yNextValue = yValue + (sigmaValue*xValue)*2**(-x)
    zNextValue = zValue - sigmaValue * atanValues[x]

    # Determining the signum of next angle (addition or subtraction)
    if zNextValue >= 0:
        sigmaNextValue = 1
    else:
        sigmaNextValue = -1

    # Values for new iteration
    xValue = xNextValue
    yValue = yNextValue
    zValue = zNextValue
    sigmaValue = sigmaNextValue

    print("iteration:", x, "xValue:", xValue, "yValue:", yValue, "zValue:", zValue, "sigmaValue:", sigmaValue, "\n")

# Calculating results by scaling the result values from CORDIC by the scalingValue which depends on number of iterations which were made
resultCos = scalingValues[x-1] * xValue
resultSin = scalingValues[x-1] * yValue

# Changing results sign based on the rotation of the initialZValueCordic
if (initialZValueCordic > 1.5707) and (initialZValueCordic < 3.141592):
    resultCos = - resultCos
    resultSin = - resultSin
elif (initialZValueCordic > 3.141592) and (initialZValueCordic < 4.7123):
    resultCos = - resultCos
    resultSin = - resultSin
elif (initialZValueCordic < 0) and (initialZValueCordic > - 1.5707):
    resultCos = resultCos
    resultSin = resultSin
elif (initialZValueCordic < -1.5707) and (initialZValueCordic > - 3.141592):
    resultCos = -resultCos
    resultSin = -resultSin
elif (initialZValueCordic < - 3.141592) and (initialZValueCordic > -
```

```
    4.7129):
    resultCos = - resultCos
    resultSin = - resultSin
elif (initialZValueCordic < - 4.7129) and (initialZValueCordic > - 2*3.141592):
    resultCos = resultCos
    resultSin = resultSin

# Calculating values based on the math library
mathResultCos = math.cos(initialZValueCordic)
mathResultSin = math.sin(initialZValueCordic)

# Calculating the error of CORDIC calculated values from the python math functions
errorCos = abs(resultCos) - abs(mathResultCos)
errorSin = abs(resultSin) - abs(mathResultSin)

# Result printing
print("*-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-*")
print("CORDIC results:")
print("cos: ", resultCos)
print("sin: ", resultSin)
print("scaleFactor: ", scalingValues[totalNumberOfIterations-1])

print("\n")

print("MATH results:")
print("cos: ", mathResultCos)
print("sin: ", mathResultSin)
```

Code 4 - 3 Python code of CORDIC implementation.

Once the Python implementation and debugging are completed, the Verilog implementation of the algorithm can safely created. Similar to the Division Unit module, as presented in *Calculating the division of fixed point numbers* section, the Data Path, Control Unit and Top Module were designed. This application-specific circuit design approach should be faster and safer than creating a custom CPU with reduced and customized ISA for performing the CORDIC algorithm.

4.3 IP Block Design

4.3.1 Top module design

The top module design of the CORDIC IP is illustrated in Figure 4 - 1. As evident, the structure closely resembles that of the Division Unit top module. When using an approach to create a customized circuit for an algorithm, the process of developing the top modules is likely to be similar, only with minor differences in used control signal, inputs and variables.

The Data Path module incorporates precalculated values in LUTs for *atanValues* and *scalingValues*. In this implementation, the value of *totalNumberOfIterations* is set to 12 , making the LUT 12x32 bits in size. It is worth noting that the previously introduced custom fixed-point format *Q32.15* is utilized.

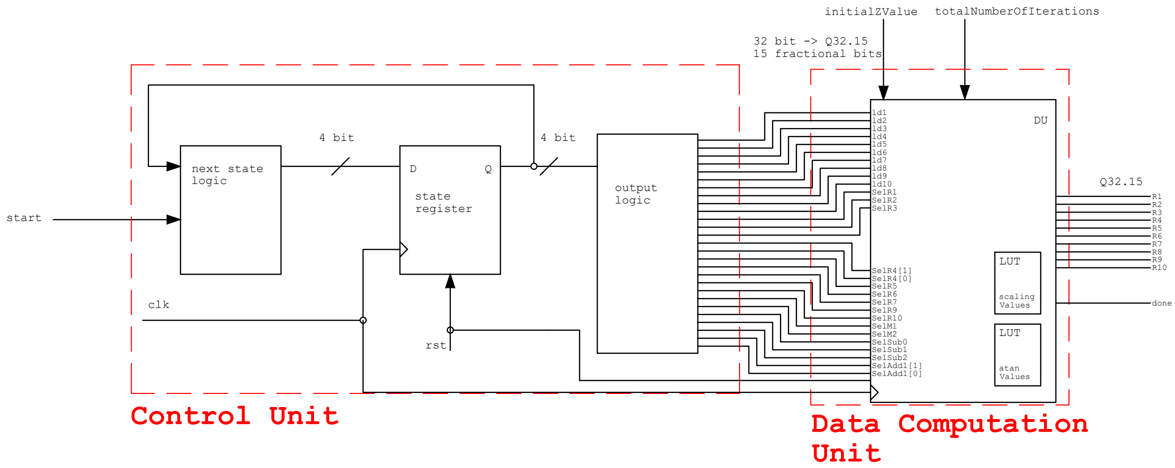


Figure 4 - 1 Top module design for the CORDIC module.

4.3.2 Allocation and Timing

In the Figure 4 - 2, the allocation and timing diagram is depicted. Notably, the `if` statements, implemented in the control unit, are documented within the diagram. The explanation, why the `if` statements are needed, is presented in the CORDIC *Theory* section.

As mentioned in the CORDIC *Control Unit* sections, there are two primary approaches to iteration cycles. The one is to proceed from *S4* to *S2* for a faster algorithm, while the other involves progressing from *S6* to *S2*. The latter approach is employed for demonstrative purposes, as it ensures that the final numerical values are always calculated.

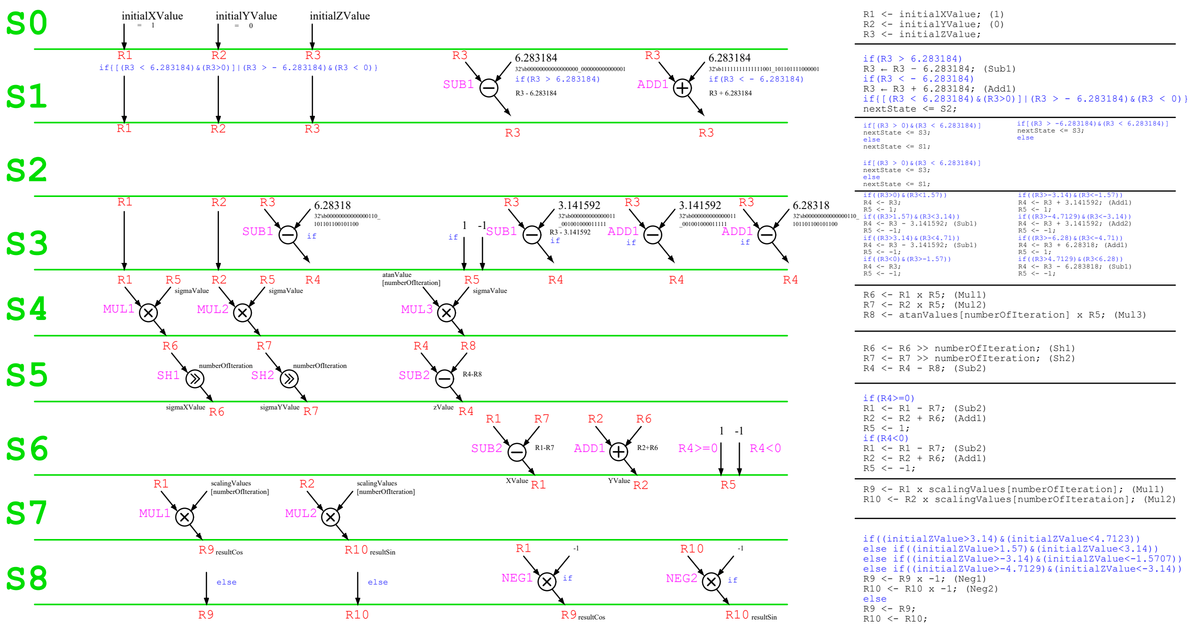


Figure 4 - 2 Allocation and timing diagram for the Data Path Unit part of the CORDIC module.

4.3.3 Data Path Module

The Figure 4 - 3 presents the Data Path module of the design with calculation and storing units included. The memory LUTs for *atanValues* and *scalingValues* are presented not as separate registers but as inputs to the calculation unit. The results of *sine* and *cosine* functions, referred to as *resultSin* and *resultCos* in the Python implementation, are stored to registers R9 and R10, respectively. It is important to note that the **NEG** blocks used for generating the negative numbers are not implemented as a standalone calculation unit blocks. Instead the negation is activated in the corresponding target register when the appropriate **SelR**$_x$ is activated. (where x represents the number of a corresponding register, either R9 or R10)

The implementation of the LUT memory module for $atanValues$ is depicted in Code 4 - 4, memory module for $scalingValues$ is depicted in Code 4 - 5.

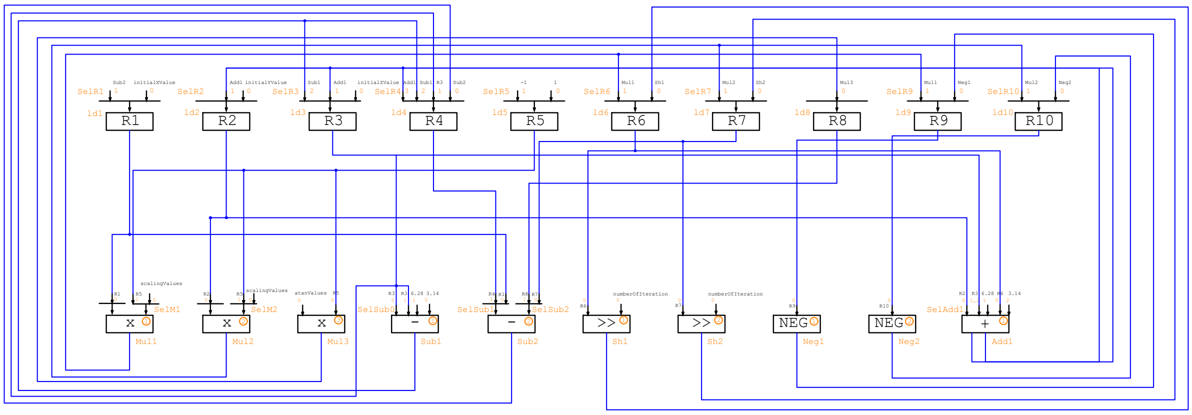


Figure 4 - 3 Register transfer level (RTL) scheme of the CORDIC module Data Path.

```
module atanValuesCordicLUT(index, returnValue);

input [3:0] index;
output reg signed [31:0] returnValue;


always@(index)
begin
    case(index)
        4'b0000: returnValue = 32'sb00000000000000000_110010010000111; // 0.7853981633974483
        4'b0001: returnValue = 32'sb00000000000000000_011101101011000; // 0.4636476090008061
        4'b0010: returnValue = 32'sb00000000000000000_001111101011011; // 0.24497866312686414
        4'b0011: returnValue = 32'sb00000000000000000_000111111101010; // 0.12435499454676144
        4'b0100: returnValue = 32'sb00000000000000000_000011111111101; // 0.06241880999595735
        4'b0101: returnValue = 32'sb00000000000000000_000001111111111; // 0.031239833430268277
        4'b0110: returnValue = 32'sb00000000000000000_000000111111111; // 0.015623728620476831
```

```
        4'b0111: returnValue = 32'sb00000000000000000_000000011111111; // 0.0078123410601011111
        4'b1000: returnValue = 32'sb00000000000000000_000000001111111; // 0.0078123410601011111
        4'b1001: returnValue = 32'sb00000000000000000_000000000111111; // 0.001953122516478818 8
        4'b1010: returnValue = 32'sb00000000000000000_000000000011111; // 0.0009765621895593195
        4'b1011: returnValue = 32'sb00000000000000000_000000000001111; // 0.00048828121119489 83
        default: returnValue = 32'sb00000000000000000_000000000000000; // 0
    endcase
end
endmodule
```

Code 4 - 4 Verilog code of the atanValuesCordicLUT lookup table (LUT) implementation.

```
module scalingValuesCordicLUT(index, returnValue);

input [3:0] index;
output reg signed [31:0] returnValue;

always@(index)
begin
    case(index)
        4'b0000: returnValue <= 32'sb00000000000000001_000000000000000; // 1
        4'b0001: returnValue <= 32'sb00000000000000000_101101010000010; // 0.7071067811865476
        4'b0010: returnValue <= 32'sb00000000000000000_101000011110100; // 0.6324555320336759
        4'b0011: returnValue <= 32'sb00000000000000000_100111010001001; // 0.6135719910778964
        4'b0100: returnValue <= 32'sb00000000000000000_100110111101110; // 0.6088339125177524
        4'b0101: returnValue <= 32'sb00000000000000000_100110111000111; // 0.6088339125177524
        4'b0110: returnValue <= 32'sb00000000000000000_100110110111101; // 0.607351770141296
        4'b0111: returnValue <= 32'sb00000000000000000_100110110111011; // 0.6072776440935261
        4'b1000: returnValue <= 32'sb00000000000000000_100110110111010; // 0.6072591122988928
        4'b1001: returnValue <= 32'sb00000000000000000_100110110111010; // 0.6072544793325625
        4'b1010: returnValue <= 32'sb00000000000000000_100110110111010; // 0.6072533210898753
        4'b1011: returnValue <= 32'sb00000000000000000_100110110111010; // 0.6072530315291345
```

```verilog
        default: returnValue <= 32'sb0000000000000000_000000000000000; //
    0
    endcase
end
endmodule
```

Code 4 - 5 Verilog code of the scalingValuesCordicLUT lookup table (LUT) implementation.

4.3.4 Control Unit

Similarly to the Division *Control Unit* section, the encoding of the control signal is presented in Table 4 - 1.

The branches of `if` statements used in the design have been color-coded to enhance clarity. Steps *S5* and *S6* are mainly focused on multiplying the result of iteration by the appropriate scaling value and on multiplying the calculated values by 1 or -1 based on the quadrant of the original reference angle.

Table 4 - 1 Control signal encoding table for instructions to be processed by the CORDIC Module.

State	RTL Code	27 ld1	26 ld2	25 ld3	24 ld4	23 ld5	22 ld6	21 ld7	20 ld8	19 ld9	18 ld10	17 SelR1	16 SelR2	15 SelR3[1]	14 SelR3[0]	13 SelR4[1]	12 SelR4[0]	11 SelR5	10 SelR6	9 SelR7	8 SelR9	7 SelR10	6 SelM1	5 SelM2	4 SelSub0	3 SelSub1	2 SelSub2	1 SelAdd1[1]	0 SelAdd[0]	CV
S0	R0 ← totalNumberOfIterations; R1 ← initialXValue; R2 ← initialYValue; R3 ← initialZValue;	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	28'hE000000
S1	if(R3 >6.283184) R3 ← R3 – 6.283184; (Sub1)	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	28'h2008010
	if(R3 <- 6.283184) R3 ← R3 + 6.283184; (Add1)	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	28'h2004001
	if{[(R3 <6.283184)&(R3>0)]|(R3 >- 6.283184)&(R3 <0)} → nextState <= S2;																													
S2	if[(R3 >0)&(R3 <6.283184)] → nextState <= S3; CS = 0; else → nextState <= S1; if[(R3 >0)&(R3 <6.283184)] → nextState <= S3; CS = 0; else → nextState <= S1; if[(R3 >-6.283184)&(R3 <6.283184)] → nextState <= S3; CS = 0; else → nextState <= S1;	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	28'h0
S3	if[(R3 >0)&(R3 <1.5707)] R4 ← R3; R5 ← 1;	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	28'h1801000
	if[(R3 >1.5707)&(R3 <3.141592)] R4 ← R3 – 3.141592; (Sub1) R5 ← -1;	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	28'h1802800
	if[(R3 >3.141592)&(R3 <4.7123)] R4 ← R3 – 3.141592; (Sub1) R5 ← 1;	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	28'h1802000
	if[(R3 <0)&(R3 >- 1.5707)] R5 ← -1; R4 ← R3;	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	28'h1801800
	if[(R3 <- 1.5707)&(R3 >- 3.141592)] R4 ← R3 + 3.141592; (Add1) R5 ← 1;	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	28'h1803002
	if[(R3 <- 3.141592)&(R3 >- 4.7129)] R5 ← -1; R4 ← R3 + 3.141592; (Add1)	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0	28'h1803802
	if[(R3 <- 4.1729)&(R3 >- 6.28318)] R5 ← 1; R4 ← R3 + 6.28318; (Add1)	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	28'h1803001
	if[(R3 >4.1729)&(R3 <6.28318)] R5 ← -1; R4 ← R3 – 6.28318; (Sub1)	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0	0	0	28'h1802810
S4	R6 ← R1 x R5; (Mul1) R7 ← R2 x R5; (Mul2) R8 ← atanValues[numberOfIteration] x R5; (Mul3)	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	0	0	0	0	0	28'h700660
S5	R6 ← R6 »numberOfIteration; (Sh1) R7 ← R7 »numberOfIteration; (Sh2) R4 ← R4 – R8; (Sub2)	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	28'h160000C
S6	R4 >= 0; R1 ← R1 – R7; (Sub2) R2 ← R2 + R6; (Add1) R5 ← 1;	1	1	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	28'hC830000
	R4 <0; R1 ← R1 – R7; (Sub2) R2 ← R2 + R6; (Add1) R5 ← -1;	1	1	0	0	1	0	0	0	0	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	28'hC830800
S7	R9 ← R1 x scalingValues[numberOfIteration]; (Mul1) R10 ← R2 x scalingValues[numberOfIteration]; (Mul2)	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	28'hC0180
S8	if[(R3 >3.141592)&(R3 <4.7123)] R9 ← R9 x (-1); (Neg1) R10 ← R10 x (-1); (Neg2)	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	28'hC0000
	if[(R3 >1.5707)&(R3<3.141592)] R9 ← R9 x (-1); (Neg1) R10 ← R10 x (-1); (Neg2)	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	28'hC0000
	if[(R3 >-3.141592)&(R3 <- 1.5707)] R9 ← R9 x (-1); (Neg1) R10 ← R10 x (-1); (Neg2)	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	28'hC0000
	if[(R3 >- 4.7129)&(R3 <- 3.141592)] R9 ← R9 x (-1); (Neg1) R10 ← R10 x (-1); (Neg2)	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	28'hC0000
	else R9 ← R9; R10 ← R10;	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	28'h0

4.4 Simulation results

The testbench for testing the design was developed using Cocotb [1] with the Verilator [2] as a simulator.

During the algorithm implementation process it becomes evident, that the number of cycles required for the final calculation can be determined as

$$NoCyc_{\text{result every iteration}} = \left\{ \begin{array}{ll} 3, & \text{if } initialZValue \in [-2\pi, 2\pi] \\ 4, & \text{if } initialZValue \notin [-2\pi, 2\pi] \end{array} \right\} + 5NoIt, \tag{4 - 11}$$

where $NoCyc$ (-) is the number of cycles and $NoIt$ is the number of iterations for the CORDIC algorithm. The 4 value is caused by states *S0*-*S4* and the multiplication by 5 is caused by iterting through states *S4*–*S8*. When the result of the CORDIC algorithm is calculated only once at the end of the algorithm, the number of iterations can be determined by

$$NoCyc_{\text{result at the end}} = \left\{ \begin{array}{l} 3,\ \text{if}\ initialZValue \in [-2\pi, 2\pi] \\ 4,\ \text{if}\ initialZValue \notin [-2\pi, 2\pi] \end{array} \right\} +\ 3NoIt + 2, \qquad (4 - 12)$$

where the multiplication by value 3 is caused by iterating through states *S4*–*S6*, the addition of 4 is caused by states *S0*-*S4* and the addition of the 2 is casued by states *S7*–*S8* calculating the final result.

In the simulation the $numberOfCycles$ displayed is an index of the cycle, so for angle $\theta = -1.247985$ rad is the number of iterations depicted on Figure 4 - 5 is 63.

The frequency of the clock signal in the simulation is currently set to 50 MHz.

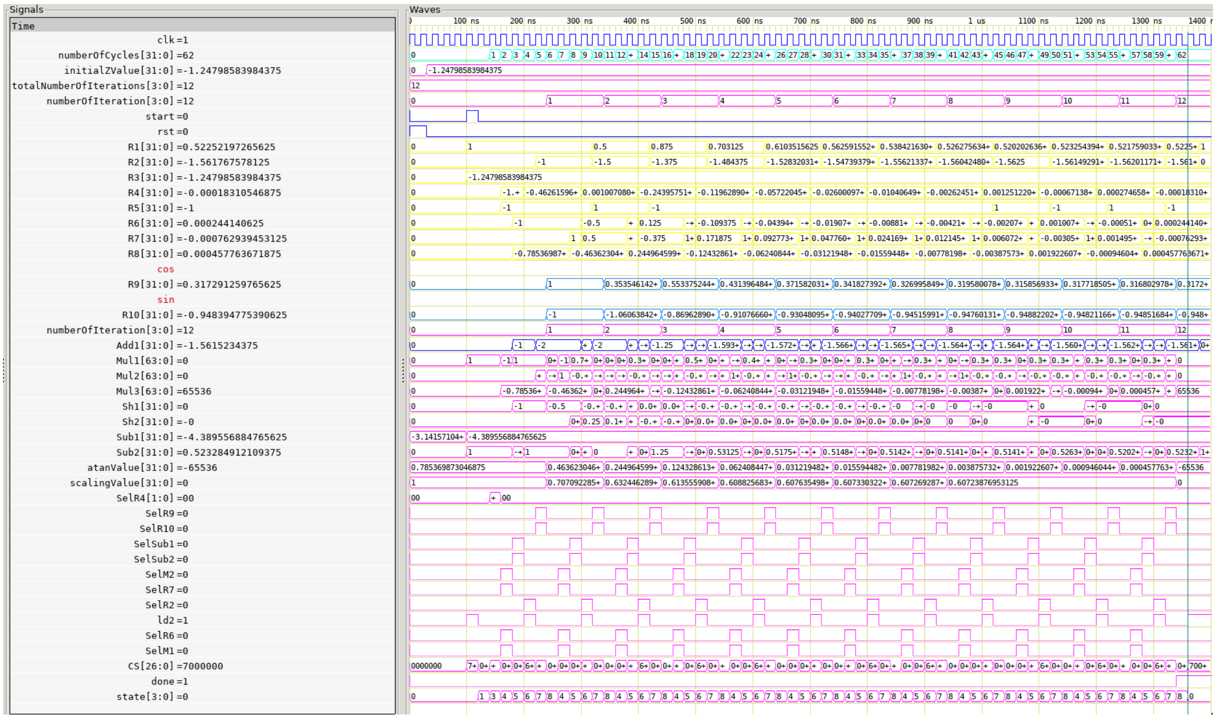


Figure 4 - 4 The Verilog simulation of CORDIC algorithm for determining the sine and cosine values of angle $\theta = -1.2479$ rad. The actual scaled value of sine and cosine is calculated every iteration with this algorithm approach. The result is passed to registers R9 and R10.

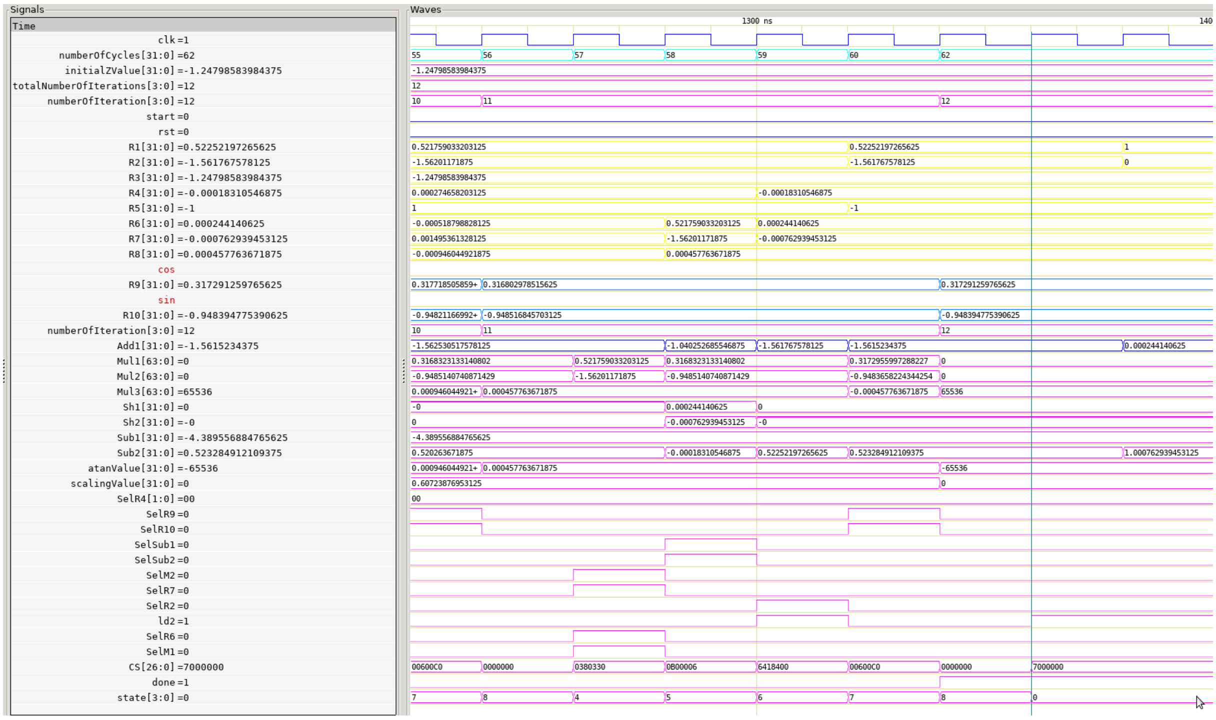


Figure 4 - 5 The detail of the last iteration of the Verilog simulation of CORDIC algorithm for determining the sine and cosine values of angle $\theta = -1.2479$ rad.The actual scaled value of sine and cosine is calculated every iteration with this algorithm approach. The result is passed to registers R9 and R10.

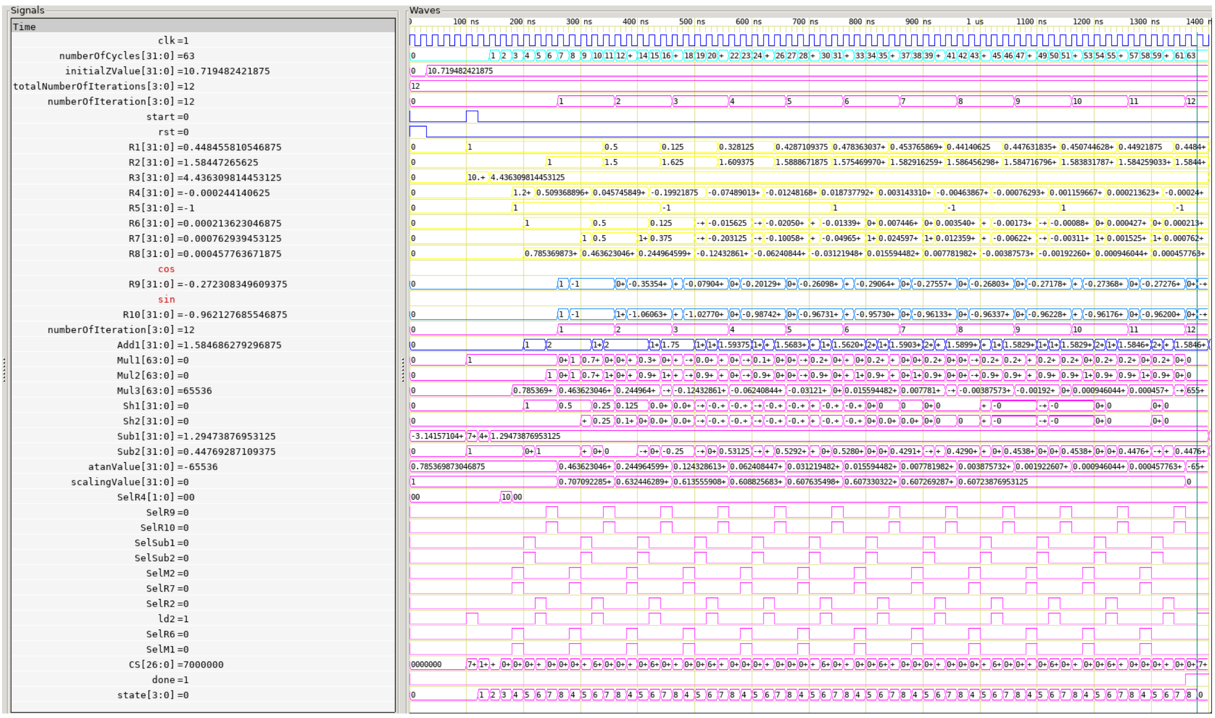


Figure 4 - 6 The Verilog simulation of CORDIC algorithm for determining the sine and cosine values of angle $\theta = 10.7195129$ rad. The actual scaled value of sine and cosine is calculated every iteration with this algorithm approach. The result is passed to registers R9 and R10.

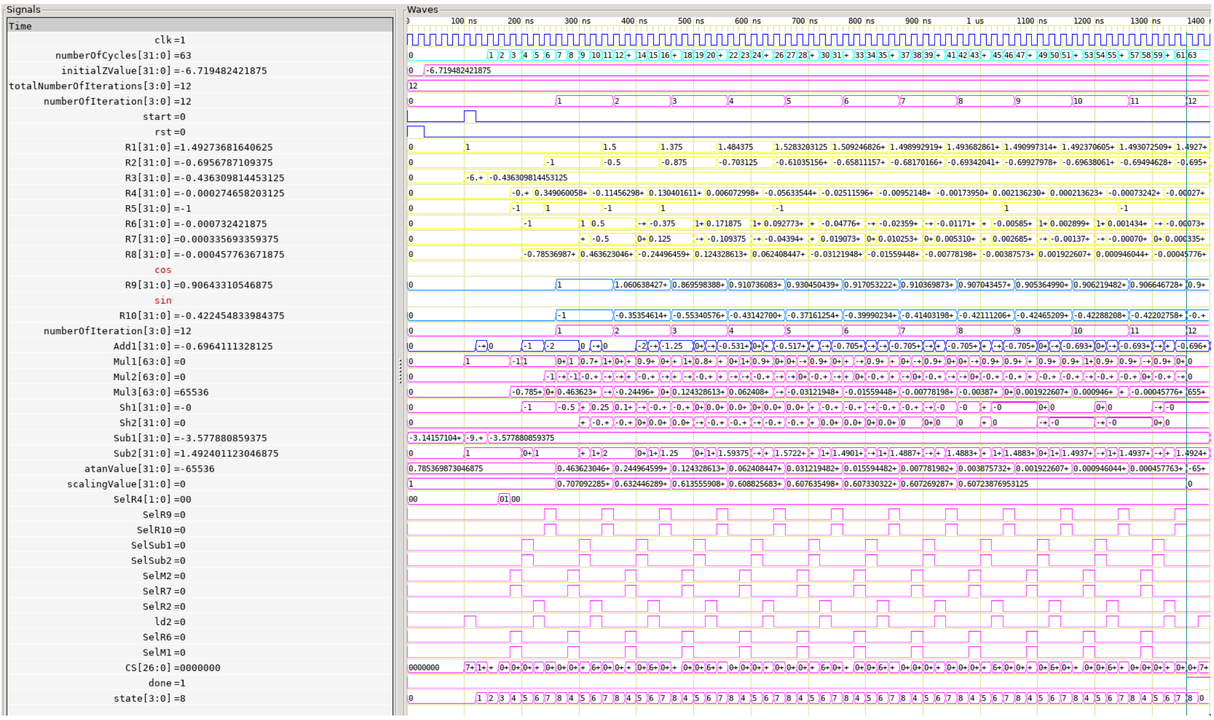


Figure 4 - 7 The Verilog simulation of CORDIC algorithm for determining the sine and cosine values of angle $\theta = -6.7195129$ rad. The value of sinus and cosinus based on the current iteration is also calculated in this algorithm approach. The result is passed to registers R9 and R10.

5 Simple set of nonlinear equations solved by a Newton-Raphson algorithm using a custom circuit implementation

Most of the modules presented in the preceding sections can be utilized as submodules to solve the system of nonlinear equations. Because this work aims to solve the transcendetal equations for Selective Harmonic Elimination (SHE), the most effective approach is to initially solve a simpler set of equations to determine the difficulty and viability of the NR.

5.1 Theory

The objective of the NR algorithm is to solve the set of nonlienar equations

$$\mathrm{F}_1(x_1, x_2) = x_1^3 - x_2 - 1, \tag{5 - 1}$$

$$\mathrm{F}_2(x_1, x_2) = x_1 - 2x_2 - 2, \tag{5 - 2}$$

where one possible set of solutions x_1 and x_2 yields

$$\mathrm{F}_1 = 0, \tag{5 - 3}$$

$$\mathrm{F}_2 = 0. \tag{5 - 4}$$

The algorithm could have been implemented in a custom CPU with reduced instruction set. However, due to apparent reasons such as speed and complexity associated with developing own processor, chosen approach involved creating an application specific circuit design.

In order to integrate the algorithm into the custom design, the general NR algorithm approach had to be simplified to the its most fundamental implementation. Every component that could be precalculated was set as a static value during the design phase.

To check if the implementation and algorithm was well designed, the solution by *Solve* function and a customized NR was made in Wolfram Mathematica.

Before initiating the algorithm, the starting values of x_1^0 and x_2^0 are set as inputs to the module. Based on that input the function values at selected starting points are calculated.

As a next step, the so called defect can be calculated using the newly found values of $\mathrm{F}_1(x_1^0, x_2^0)$ and $\mathrm{F}_2(x_1^0, x_2^0)$

$$\Delta\mathbf{F}^i = \begin{pmatrix} \Delta\mathrm{F}_1^i \\ \Delta\mathrm{F}_2^i \end{pmatrix} = \begin{pmatrix} \mathrm{F}_1^i - \mathrm{F}_1^{\text{known solution}} \\ \mathrm{F}_2^i - \mathrm{F}_2^{\text{known solution}} \end{pmatrix}, \tag{5 - 5}$$

where the superscript i is the number of iteration for which the defect is calculated. When the algorithm starts, the $i = 0$. So for example the input value for F_1^0 is x_1^0 and x_2^0.

Next, the Jacobian matrix **J** from vector of functions $(\boldsymbol{F})(x_1, x_2) = (\mathrm{F}_1, \mathrm{F}_2)$ is calculated as follows.

$$\mathbf{J}^i = \begin{pmatrix} \frac{\mathrm{dF}_1}{\mathrm{d}x_1^i} & \frac{\mathrm{dF}_1}{\mathrm{d}x_2^i} \\ \frac{\mathrm{dF}_2}{\mathrm{d}x_1^i} & \frac{\mathrm{dF}_2}{\mathrm{d}x_2^i} \end{pmatrix} = \begin{pmatrix} 3(x_1^i)^2 & -1 \\ 1 & -2 \end{pmatrix}. \tag{5 - 6}$$

As for the general NR algorithm, the inverted value Jacobian matrix needs to be calculated. The problem is, that when using general mathematical software, such as Wolfram Mathematica, the calculation of the inversion is as easy as using function of inversion. When designing the circuit, the approach of manual calculation of inversion must be used. In this paper, the calculation is made possible by calculating the determinant of the Jacobian Matrix, its reciprocal value, adjugate matrix of the Jacobian Matrix and multiplication of the adjugate matrix elements by the calculated determinant reciprocal value.

Because the size of the Jacobian matrix is 2x2 the determinant may be easily calculated using the Sarrus Rule. When the matrix is more complicated, the expansion method may be utilized.

$$\det(\mathbf{J}) = 3(x_1^i)^2(-2) - (-1) = 3(x_1^i)^2(-2) + 1. \tag{5 - 7}$$

The reciprocal value of the determinant is then calculated by the Division Unit, created for calculating division of arbitrary real numbers. This Division Unit is presented in the section *Calculating the division of fixed point numbers*.

The adjugate matrix is calculated as follows

$$\operatorname{adj}(\mathbf{J}) = \begin{pmatrix} \mathbf{J}_{11}(-1)^{1+1} & \mathbf{J}_{01}(-1)^{1+2} \\ \mathbf{J}_{10}(-1)^{1+2} & \mathbf{J}_{00}(-1)^{2+2} \end{pmatrix} = \begin{pmatrix} -2 & -1 \\ 1 & 3(x_1^i)^2 \end{pmatrix}. \tag{5 - 8}$$

After the calculation of the reciprocal value of the determinant of the Jacobian matrix and the adjugate matrix, the inverted Jacobian matrix may be finally calculated

$$\mathbf{J}^{-1i} = \frac{1}{\det(\mathbf{J}^i)} \begin{pmatrix} \operatorname{adj}(\mathbf{J}_{00}^i) & \operatorname{adj}(\mathbf{J}_{01}^i) \\ \operatorname{adj}(\mathbf{J}_{10}^i) & \operatorname{adj}(\mathbf{J}_{10}^i) \end{pmatrix} = \frac{1}{\det(\mathbf{J}^i)} \begin{pmatrix} -2 & -1 \\ 1 & 3(x_1^i)^2 \end{pmatrix}. \tag{5 - 9}$$

Next the $(\Delta x_1^i, \Delta x_2^i)$ can be calculated using the inverted Jacobian matrix and the defect.

$$\begin{pmatrix} \Delta x_1^i \\ \Delta x_2^i \end{pmatrix} = \begin{pmatrix} \mathbf{J}_{00}^{-1,i}\ \Delta \mathrm{F}_1^i + \mathbf{J}_{01}^{-1,i}\ \Delta \mathrm{F}_2^i \\ \mathbf{J}_{10}^{-1,i}\ \Delta \mathrm{F}_1^i + \mathbf{J}_{11}^{-1,i}\ \Delta \mathrm{F}_2^i \end{pmatrix}. \tag{5 - 10}$$

Now the next iteration value denoted as $i + 1$ of x_1 and x_2 may be calculated

$$\begin{pmatrix} x_1^{i+1} \\ x_2^{i+1} \end{pmatrix} = \begin{pmatrix} x_1^i + \Delta x_1^i \\ x_2^i + \Delta x_2^i \end{pmatrix}. \tag{5 - 11}$$

With these new iteration values x_1^{i+1} x_2^{i+1} the loop for calculation starts again at the calculation of new values F_1^{i+1} and F_2^{i+1} which is presented at the start of this section.

5.2 IP Block Design

5.2.1 Top module design

Figure 5 - 1 depicts the top module design of the circuit. The Control Unit sends control signals to the Data Path unit to make the calculations. As in all designs in this paper, the numbers are formatted in the *Q32.15* fixed point format.

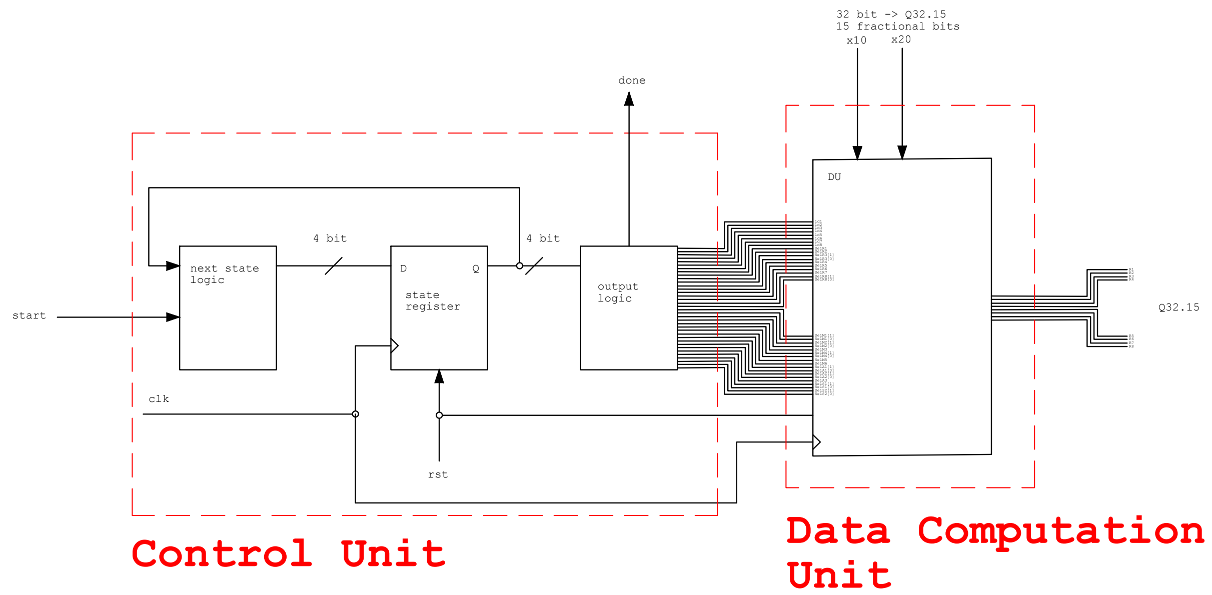


Figure 5 - 1 Top module design for the simple Newton-Raphson (NR) calculation module block.

5.2.2 Allocation and Timing

The algorithm structure for the Verilog implementation is depicted in the data flow diagram in the Figure 5 - 2.

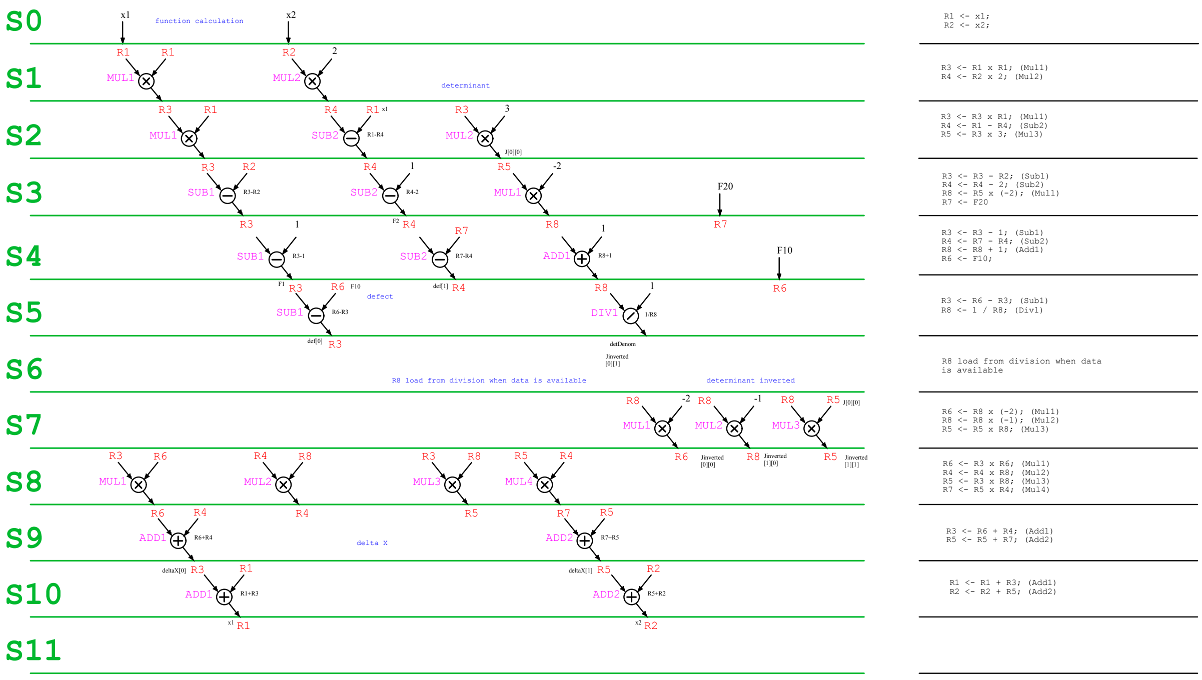


Figure 5 - 2 Allocation and timing diagram for the Data Path of the simple Newton-Raphson (NR) module.

5.2.3 Data Path Unit

The Data path unit for this simple NR algorithm consists of four multipliers, two adders, two subtractors and one divider. The divider is implemented using the Division Unit, presented in the section *Calculating the division of fixed point numbers*. Upon completion of the algorithm the results for x_1 and x_2 are saved in the R1 and R2 registers, the state transitions to *S11* and signal *done* is set to 1. The results then can be driven to another module or unit for further usage.

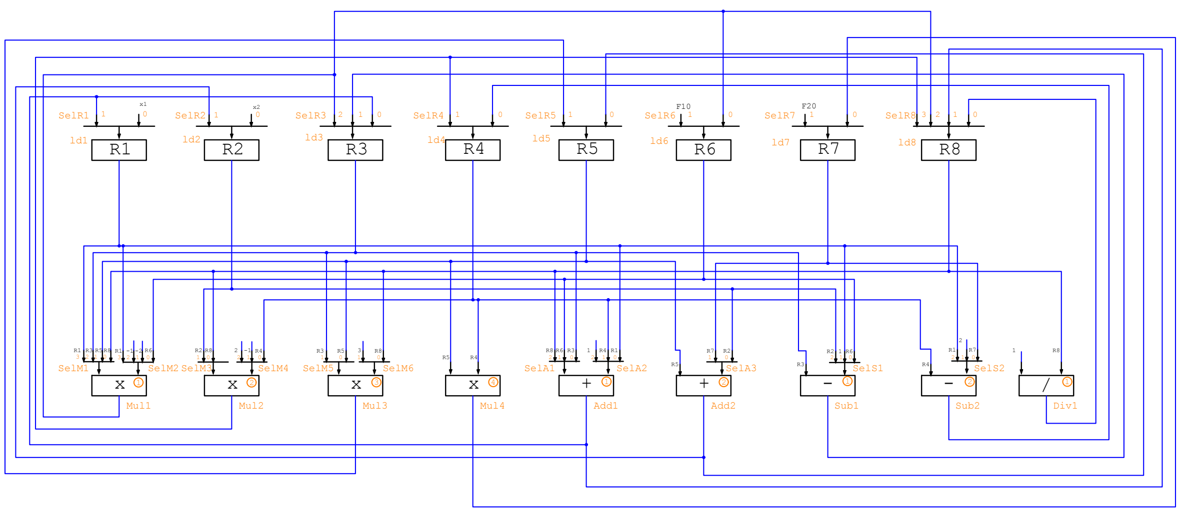


Figure 5 - 3 Register Transfer Level (RTL) scheme of the Data Path part of the simple Newton-Raphson (NR) calculation module.

5.2.4 Control Unit

The Table 5 - 1 shows encoding of a control signal for the Data Path unit.

The NR algorithm iteration transitions from the state *S10* to state *S1* when the iteration count is lower than the predetermined total number of iterations, value which is set in the Control Unit during the design phase. In this particular implementation, the total number of iterations is set to 5. It is worth noting that sometimes the termination of the NR algorithm is determined by the value of a defect. However, in this implementation the defect-check is not implemented.

Implementation of a defect-controlled algorithm would be straightforward. The values from registers holding the defect values, R3 and R4, would be connected to the control unit in the steps *S4* and *S5* respectively, and a comparison with the reference defect value would be executed. If the defect value was smaller than the reference value, the algorithm would transition to the state *S11* and therefore the calculation would end. Conversely, if the defect was larger than the reference value, the next state would be *S6* and the iteratioun would proceeed normally, transitioning from state *S10* to *S1*.

Table 5 - 1 Control signal encoding table for instructions to be processed by the simple Newton-Raphson (NR) alogrithm Module.

State	RTL Code	35 ld1	34 ld2	33 ld3	32 ld4	31 ld5	30 ld6	29 ld7	28 ld8	27 SelR1	26 SelR2	25 SelR3[1]	24 SelR3[0]	23 SelR4	22 SelR5	21 SelR6	20 SelR7	19 SelR8[1]	18 SelR8[0]
S0	R1 ← x1; R2 ← x2;	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S1	R3 ← R1 x R1; (1) R4 ← R2 x 2; (2)	0	0	1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0
S2	R3 ← R3 x R1; (1) R4 ← R1 – R4; (2) R5 ←R3 x 3; (3)	0	0	1	1	1	0	0	0	0	0	1	0	0	1	0	0	0	0
S3	R3 ← R3 – R2; (1) R4 ← R4 – 2; (2) R8 ← R5 x (-2); (1) R7 ← F20;	0	0	1	1	0	0	1	1	0	0	0	1	0	0	0	1	1	0
S4	R3 ← R3 – 1; (1) R4 ← R7 – R4; (2) R8 ← R8 + 1; (1) R6 ← F10;	0	0	1	1	0	1	0	1	0	0	0	1	0	0	1	0	0	1
S5	R3 ← R6 – R3; (1) R8 ← 1 / R8; (1)	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0
S6	R8 load from division when data is available	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
S7	R6 ← R8 x (-2); (1) R8 ← R8 x (-1); (2) R5 ← R5 x R8; (3)	0	0	0	0	1	1	0	1	0	0	0	0	0	1	0	0	1	1
S8	R6 ← R3 x R6; (1) R4 ← R4 x R8; (2) R5 ← R3 x R8; (3) R7 ← R5 x R4; (4)	0	0	0	1	1	1	1	0	0	0	0	0	1	1	0	0	0	0
S9	R3 ← R6 + R4; (1) R5 ← R5 + R7; (2)	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
S10	R1 ← R1 + R3; (1) R2 ← R2 + R5; (2)	1	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
S11		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

State	RTL Code	17 SelM1[1]	16 SelM1[0]	15 SelM2[1]	14 SelM2[0]	13 SelM3	12 SelM4[1]	11 SelM4[0]	10 SelM5	9 SelM6	8 SelA1[1]	7 SelA1[0]	6 SelA2[1]	5 SelA2[0]	4 SelA3	3 SelS1[1]	2 SelS1[0]	1 SelS2[1]	0 SelS2[0]	CS
S0	R1 ← x1; R2 ← x2;	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	36’hC00000000
S1	R3 ← R1 x R1; (1) R4 ← R2 x 2; (2)	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	36’h30283F000h
S2	R3 ← R3 x R1; (1) R4 ← R1 – R4; (2) R5 ←R3 x 3; (3)	1	0	1	1	0	0	0	1	1	0	0	0	0	0	0	0	1	0	36’h38242C602
S3	R3 ← R3 – R2; (1) R4 ← R4 – 2; (2) R8 ← R5 x (-2); (1) R7 ← F20;	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	36’h331198009
S4	R3 ← R3 – 1; (1) R4 ← R7 – R4; (2) R8 ← R8 + 1; (1) R6 ← F10;	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	1	0	0	36’h351240144
S5	R3 ← R6 – R3; (1) R8 ← 1 / R8; (1)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	36’h211000000
S6	R8 load from division when data is available	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	36’h10000000
S7	R6 ← R8 x (-2); (1) R8 ← R8 x (-1); (2) R5 ← R5 x R8; (3)	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	36’hD04C4800
S8	R6 ← R3 x R6; (1) R4 ← R4 x R8; (2) R5 ← R3 x R8; (3) R7 ← R5 x R4; (4)	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	36’h1E0C20400
S9	R3 ← R6 + R4; (1) R5 ← R5 + R7; (2)	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	0	36’h2800000B0
S10	R1 ← R1 + R3; (1) R2 ← R2 + R5; (2)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	36’hC0C000000
S11		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	36’hxxxxxxxxx

5.3 Simulation results

The test bench for simulation was made using Cocotb [1] with the Verilator [2] as a simulator. The results of the calculation may be seen in the registers R1 and R2. The results are $x_1 = -0.707489$ and $x_2 = -1.353759$.

The clock signal frequency in simulation was set to 20 MHz.

Figure 5 - 4 The complete Verilog simulation of a simple Newton-Raphson (NR) algorithm. The result may be seen in registers R1 and R2 after the fifth iteration of the algorithm.

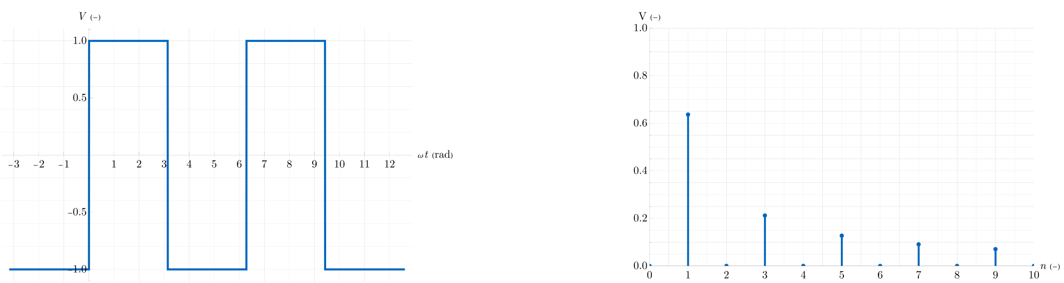
6 Selective Harmonic Elimination

6.1 Theory

The original theory for Selective Harmonic Elimination was initially developed in [8, 9] and later adopted by numerous researchers for various voltage inverter topologies. Currently, the strategy is primarily employed in traction applications after start up state ends and the reference voltage for the drive is high enough so the six step output voltage is utilized. However, the general six step output signal produces high-order harmonics. When the motor is powered by these high-order voltage harmonics, the current with high-order harmonics (excluding triplen harmonics, considering the symmetric 3 phase motor) is observed. These current harmonics result in undesirable current ripple, torque ripple and losses [10], thereby decreasing the efficiency of the drive.

To control the output voltage and reduce unwanted harmonics, the Selective Harmonic Elimination (SHE) technique can be employed. The elimination is based on generating the output voltage by switching components at certain phase angles, thereby generating waveform with a number of pulses, to corresponding the number of elliminated harmonics. The calculation which angles to use is based on the of Fourier coefficients theory. Equations describing the switching angles have been adapted for different types of converters, including multilevel, H-bridge converters or generic Voltage Source Inverters (VSI). In this paper, the regular two level VSI is considered.

The considered inverter phase voltage Six-Step output waveform is depicted in Figure 6 - 1a, while the harmonic analysis of the generic waveform is depicted in Figure 6 - 1b. It's worth noting that in a three-phase symmetrical system, the triplen harmonics are also eliminated.



(a) Generic Six-Step Waveform output of a two level Voltage Source Inverter. The Voltage value is normalized to a DC link voltage.

(b) Generic Six-Step Waveform harmonics analysis. The Voltage value is normalized to a DC link voltage.

Figure 6 - 1 Six-Step voltage waveform and harmonic analysis.

As previously mentioned, the SHE method is based on a Fourier coefficient analysis. When the odd quarter-wave symmetry of the waveform is assumend, the a_n Fourier coefficient is zero (as mentioned in the Equation 6 - 1), whereas the b_n coefficient may be written as Equation 6 - 2.

$$a_n = 0, \tag{6 - 1}$$

$$b_n = \frac{2}{T}\int_0^T x(n\omega t)\sin(\omega t)\mathrm{d}\omega t, \tag{6 - 2}$$

where the T is signal periode, $x(\omega t)$ description of the VSI output voltage waveform and n is the order of the harmonics.

When assuming quarter-wave symmetry the Equation 6 - 2 may be rewritten as

$$b_n = \frac{8}{T}\int_0^{T/4} x(\omega t)\sin(n\omega t)\mathrm{d}\omega t = \frac{8}{2\pi}\int_0^{2\pi/4} x(\omega t)\sin(n\omega t)\mathrm{d}\omega t = \frac{4}{\pi}\int_0^{\frac{\pi}{2}} x(\omega t)\sin(n\omega t)\mathrm{d}\omega t. \tag{6 - 3}$$

The function $x(\omega t)$ represents the normalized output voltage pulse in relation to a DC link voltage. The Equation 6 - 2 can be reformulated by substituting ωt with the angle α, which also characterizes the output waveform in terms of radians. The function $x(\alpha)$ yields 1 when the output voltage pulse is positive and -1 when negative. The reformulated Equation 6 - 2, assuming quarter-wave symmetry, is then as follows:

$$b_n = \sum_{k=1}^{M} \frac{8}{T}\int_{\alpha_k}^{\alpha_{k+1}} x(\alpha)\sin(n\alpha)\mathrm{d}\alpha. \tag{6 - 4}$$

Here M represents number of pulses in half period of the output signal. Assuming that the integral is calculated for angles where $x(\alpha_k)$ is either 1 or -1, the function may be replaced by a constant. As a result, when $x(\alpha_k) = 1$ the integral calculation becomes straightforward.

$$b_n = \frac{4}{\pi}\sum_{k=1}^{M}\frac{1}{n}\left[-\cos(n\alpha)\right]_{\alpha_k}^{n\alpha_{k+1}} = \frac{4}{\pi n}\sum_{k=1}^{M}\left[\cos(n\alpha_k) - \cos(n\alpha_{k-1})\right]. \tag{6 - 5}$$

The Equation 6 - 5 can be further simplified by observing the results of the summation for $M = 2$.

$$\begin{aligned} b_n = \frac{4}{\pi n}\sum_{k=1}^{2}\left[\cos(n\alpha_k) - \cos(n\alpha_{k-1})\right] = \frac{4}{\pi n}\left[(\cos(n\alpha_1) - \cos(n\alpha_2)) + (\cos(n\alpha_2) - \cos(n\alpha_3))\right] = \\ = \frac{4}{\pi n}(\cos(n\alpha_1) - \cos(n\alpha_3)). \end{aligned} \tag{6 - 6}$$

According to [8] and the example calculation for $M = 2$, the further simplification of the Equation 6 - 5 is Equation 6 - 7.

$$b_n = \frac{4}{\pi n}\sum_{k=1}^{M}(-1)^{k+1}\cos(n\alpha_k). \tag{6 - 7}$$

It can be said, that the number of eliminated odd harmonics is $N = M - 1$.

To maintain clarity of this paper only the 5th harmonics is being eliminated by the designed unit. The set of equations when incorporating the voltage level V_{DC} in a VSI DC link required to eliminate this harmonic is as follows.

$$\begin{aligned} V_1 = b_1 = V_{\mathrm{DC}}\frac{4}{\pi}\left[\cos(\alpha_1) - \cos(\alpha_2)\right], \\ V_5 = b_5 = V_{\mathrm{DC}}\frac{4}{5\pi}\left[\cos(5\alpha_1) - \cos(5\alpha_2)\right]. \end{aligned} \tag{6 - 8}$$

The amplitudes of the 1st and 5th harmonics are denoted as $V_1 = b_1$ and $V_5 = b_5$, respectively. For the elimination of the 5th harmonic, it is required that $b_5 = 0$. Consequentely, the set of Equations 6 - 8 can be simplified as set of Equations 6 - 9.

$$
\begin{aligned}
\frac{\pi V_1}{4V_{\mathrm{DC}}} &= \cos(\alpha_1) - \cos(\alpha_2), \\
0 &= \cos(5\alpha_1) - \cos(5\alpha_2).
\end{aligned} \tag{6 - 9}
$$

Solving the nonlinear Equations 6 - 9 is not straightforward. Various methods can be employed for solving the problem, such as Genetic Algorithms [11, 12, 13] or algebraic methods [14, 15]. One commonly used algebraic method is Newton-Raphson (NR) algorithm [16]. In this paper, the solution is obtained solely using NR algorithm. However, it's worth noting that the success of this method depends on setting the initial conditions correctly; otherwise, a solution may not be found. In contrast, Genetic Algorithms also require setting initial values, but they often use random numbers from predefined intervals.

In real-time systems, the approach for solving the SHE equations may often be to precalculate the required switching angles offline and the utilize the LUT in a microprocessor to determine which set of angles use for the set reference voltage. Nowadays the FPGAs are more frequently utilized to calculate the solution. The calculation can be highly paralelized and optimized, enabling the solution to be obtained in near real-time. In the following sections the prototype implementation in Python and final implementaion in Verilog are presented.

6.2 Simplification for Verilog and High level implementation

When implementing the solution in computational software like Wolfram Mathematica, optimizing the algorithm is unnecessary. However, when implementing the algorithm to an FPGA, higher-level constructs are not automatically available, so the simplification is necessary. Before creating the Verilog design, it is suitable, for clarity and prototyping purposes, to implement the algorithm in Python. In this section, the simplified algorithm of a NR aglorithm is presented.

The set of equations for eliminating the 5th harmonics may be formulated as

$$
\begin{aligned}
\mathrm{F}_1^i &= \cos(\alpha_1) - \cos(\alpha_2), \\
\mathrm{F}_2^i &= \cos(5\alpha_1) - \cos(5\alpha_2), \\
&\text{where } \mathrm{F}_1^0 = m\,\frac{\pi}{4},\ \mathrm{F}_2^0 = 0.
\end{aligned} \tag{6 - 10}
$$

Where $m = V_1/V_{\mathrm{DC}}$ is modulation index.

Thus the Jacobian matrix is

$$
\mathbf{J}^i = \begin{pmatrix} -\sin(\alpha_1^i) & \sin(\alpha_2^i) \\ -5\sin(5\alpha_1^i) & 5\sin(5\alpha_2^i) \end{pmatrix}. \tag{6 - 11}
$$

Where i is the index of the iteration of the algorithm. Also the inverted Jacobian matrix is needed for further calculations.

$$
\mathrm{J}^{-1,i} = \begin{pmatrix} \frac{5\sin(5\alpha_2^i)}{5\sin(5\alpha_1^i)\sin(\alpha_2^i) - 5\sin(\alpha_1^i)\sin(\alpha_2^i)} & -\frac{\sin(\alpha_2^i)}{5\sin(5\alpha_1^i)\sin(\alpha_2^i) - 5\sin(\alpha_1^i)\sin(\alpha_2^i)} \\ \frac{5\sin(\alpha_1^i)}{5\sin(5\alpha_1^i)\sin(\alpha_2^i) - 5\sin(\alpha_1^i)\sin(\alpha_2^i)} & -\frac{\sin(\alpha_1^i)}{5\sin(5\alpha_1^i)\sin(\alpha_2^i) - 5\sin(\alpha_1^i)\sin(\alpha_2^i)} \end{pmatrix}. \tag{6 - 12}
$$

From the inverted Jacobian matrix in Equation 6 - 12, it is evident that it can be easily calculated by dividing corresponding components of Jacobian matrix by the determinant, expressed as

$$
\det(\mathbf{J}) = 5\sin(5\alpha_1^i)\sin(\alpha_2^i) - 5\sin(\alpha_1^i)\sin(\alpha_2^i). \tag{6 - 13}
$$

To avoid multiple division units, in which another NR algorithm is performed, the Verilog implementation takes advantage of calculating the value $1/\det(\mathbf{J})$ with one division unit and then multiplying the inverted Jacobian matrix elements with it. This solution saves a lot of clock cycles, thus makes the algorithm faster.

Next, the defect $\Delta\mathrm{F}^i$ can be calculated

$$\begin{aligned} \Delta\mathrm{F}_1^i &= F_1^0 - F_1^i, \\ \Delta\mathrm{F}_2^i &= F_2^0 - F_2^i. \end{aligned} \tag{6 - 14}$$

After the successfully calculated defect of a current iteration, the $\Delta\alpha^i$ may be calculated.

$$\Delta\boldsymbol{\alpha}^i = \mathbf{J}^{-1,i}\Delta\boldsymbol{F}^i, \tag{6 - 15}$$

thus rewritten in components notation which is more suitable for the Verilog implementation

$$\begin{aligned} \Delta\alpha_1^i &= \mathbf{J}_{00}^{-1,i}\Delta F_1^i + \mathbf{J}_{01}^{-1,i}\Delta F_2^i, \\ \Delta\alpha_1^2 &= \mathbf{J}_{10}^{-1,i}\Delta F_1^i + \mathbf{J}_{11}^{-1,i}\Delta F_2^i. \end{aligned} \tag{6 - 16}$$

Finally the next iteration values of α_1^i and α_2^i may be calculated

$$\begin{aligned} \alpha_1^{i+1} &= \alpha_1^i + \Delta\alpha_1^i, \\ \alpha_2^{i+1} &= \alpha_2^i + \Delta\alpha_2^i. \end{aligned} \tag{6 - 17}$$

With the newly calculated values of α_1^i, α_2^i the algorithm may proceed with a new iteration $(i+1)$ for calculating the F_1^{i+1} and F_2^{i+1} values.

It is important to note, that for the NR algorithm to function correctly and yield viable results, suitable initial values F_1^0 and F_2^0 must be carefully chosen before the algorithm starts.

When elliminating the 5th harmonic with $m = 1$, the initial values of $F_2^0 = 0.08726$ rad and $F_2^0 = 1.3439$ rad yield satisfactory results.

The presented mathematical algorithm can then be transformed into an FPGA designed Verilog algorithm, visually represented as a block diagram in the section *Algorithm Block Design*.

6.3 High level implementation

The script allows changing the modulation index m at the beginning of the Python simulation. This feature enables generation of values that can be compared with results obtained from Verilog cocotb and Verilator simulation of the hardware-implemented algorithm.

The script may be run with command "*python3 she.py -mi <number>*", where *<number>* is the requested modulation index.

```python
import math
import argparse  # for parsing command line arguments

# colorama for colors, easier than init class, maybe later
# source: https://github.com/tartley/colorama
from colorama import init as colorama_init
from colorama import Fore
from colorama import Style
```

```python

colorama_init(autoreset=True)  # autoreset color on new line

# class with additional styles
class style:
    BOLD = '\033[1m'
    UNDERLINE = '\033[4m'
    END = '\033[0m'

argParser = argparse.ArgumentParser()  # new object
argParser.add_argument("-mi", "--modulationIndex", help="set the modulation
    index 0-1") # adding argument
args = argParser.parse_args()  # parsing args
modulationIndex = args.modulationIndex

# Set the desired modulation index
if not modulationIndex:
    print()
    print(style.BOLD+Fore.RED + "You did not specify the modulation index
    with mi command, specify it now:\n" + style.END)
    modulationIndex = input()

print("You have specified the modulation index: " + modulationIndex + ".\n"
    )

modulationIndex = float(modulationIndex)
totalNumberOfIterations = 10
f10 = modulationIndex * 0.7853981  # modulationIndex * pi/4
f20 = 0
x10 = 0.0872664  # 5 degree
x20 = 1.3439035  # 77 degree

x1 = x10
x2 = x20

# main NR-LOOP
for numberOfIteration in range(totalNumberOfIterations):
    prepDeltaF1 = math.cos(x1) - math.cos(x2)
    deltaF1 = f10 - prepDeltaF1

    prepDeltaF2 = math.cos(5*x1) - math.cos(5*x2)
    deltaF2 = f20 - prepDeltaF2

    prepJ11 = math.sin(x1)
    prepJ01 = math.sin(x2)
    prepJ10 = 5 * math.sin(5*x1)
    prepJ00 = 5 * math.sin(5*x2)

```

```

    prepDet1 = prepJ10 * prepJ01
    prepDet2 = 5 * prepJ11 * math.sin(5*x2)

    prepDet = prepDet1 - prepDet2

    divDet = 1 / prepDet

    jInv00 = divDet * prepJ00
    jInv01 = divDet * - prepJ01
    jInv10 = divDet * prepJ10
    jInv11 = divDet * - prepJ11


    deltaX1 = (jInv00 * deltaF1) + (jInv01 * deltaF2)
    deltaX2 = (jInv10 * deltaF1) + (jInv11 * deltaF2)

    x1 = x1 + deltaX1
    x2 = x2 + deltaX2

    print(Fore.CYAN + "numberOfIteration: " + str(numberOfIteration) +
   style.END)
# End of the main NR-LOOP

print(Fore.GREEN + "x1: " + str(x1) + style.END)
print(Fore.GREEN + "x2: " + str(x2) + style.END)
```

Code 6 - 1 Python implementation of the Selective Harmonic Elimination Algorithm with adjustable modulation index.

6.4 IP Block Design

6.4.1 Algorithm Block Diagram

The Figure 6 - 2 presents the hardware-implementation for SHE algorithm, mathematically expressed in the section *Simplification for Verilog and High level implementation*.

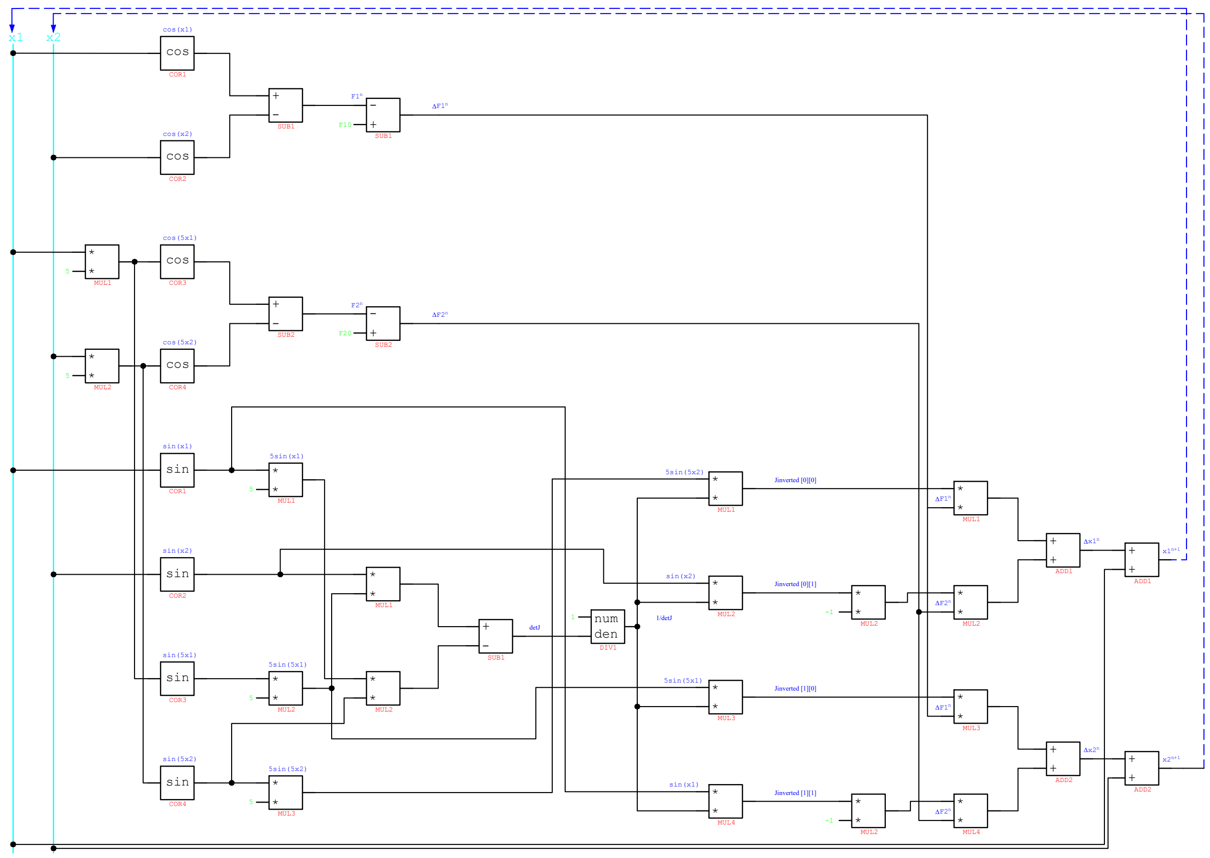


Figure 6 - 2 Block Diagram of the Selective Harmonic Elimination (SHE) algorithm using Newton-Raphson algorithm (NR). Design suitable for hardware implementation.

6.4.2 Top module design

The top module of this IP closely resembles other developed modules in this paper. The design consists of a Control Unit which sends control signals to the Data Unit. The Data Unit, which includes registers and computational units, incorporates few external sub-modules for additional calculations, such as CORDIC and division.

Consistent with every design presented, the units utilize the *Q32.15* fixed point format for the computational units and registers. The exception is the multiplier computational units, which, by the principle of multiplication, use the *Q64.30* format for results. When the multiplication results are transfered to registers, the values are rounded back to the globally used format.

The design is depicted in Figure 6 - 3.

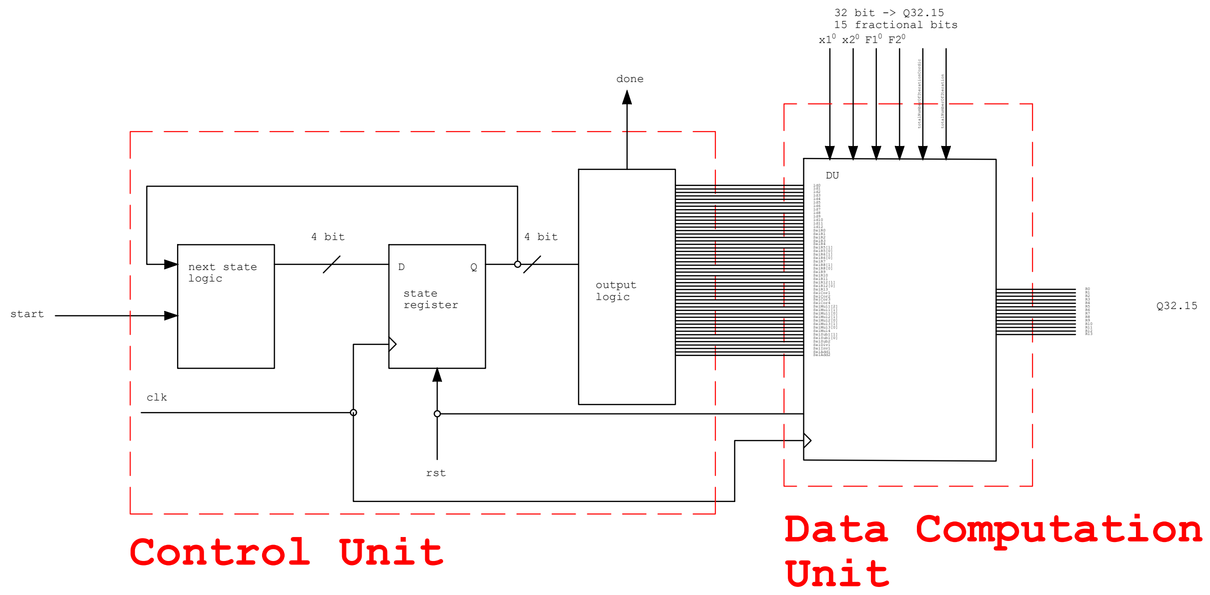


Figure 6 - 3 Top module design for the Selective Harmonic Elimination (SHE) module.

6.4.3 Allocation and Timing

The Allocation and Timing diagram, depicted in the Figure 6 - 4, outlines the algorithm presented in the *Theory* section. As evident from previous sections, this algorithm has been thoroughly tested before the Verilog implementation.

The Verilog implementation comprises a total of 13 states, labeled *S0-S12*. Through states *S1-S11*, the NR algorithm iterates to calculate the final results. The state *S0* is a starting state after resetting the unit, and state *S12* is the ending state reached after the successful calculation of the last algorithm iteration.

As previously stated, the SHE calculation module consists of various submodules, which may use other iterative algorithms. Iterations of these submodule algorithms are not in focus of this section and are implicitly accepted as a part of the SHE module algorithm.

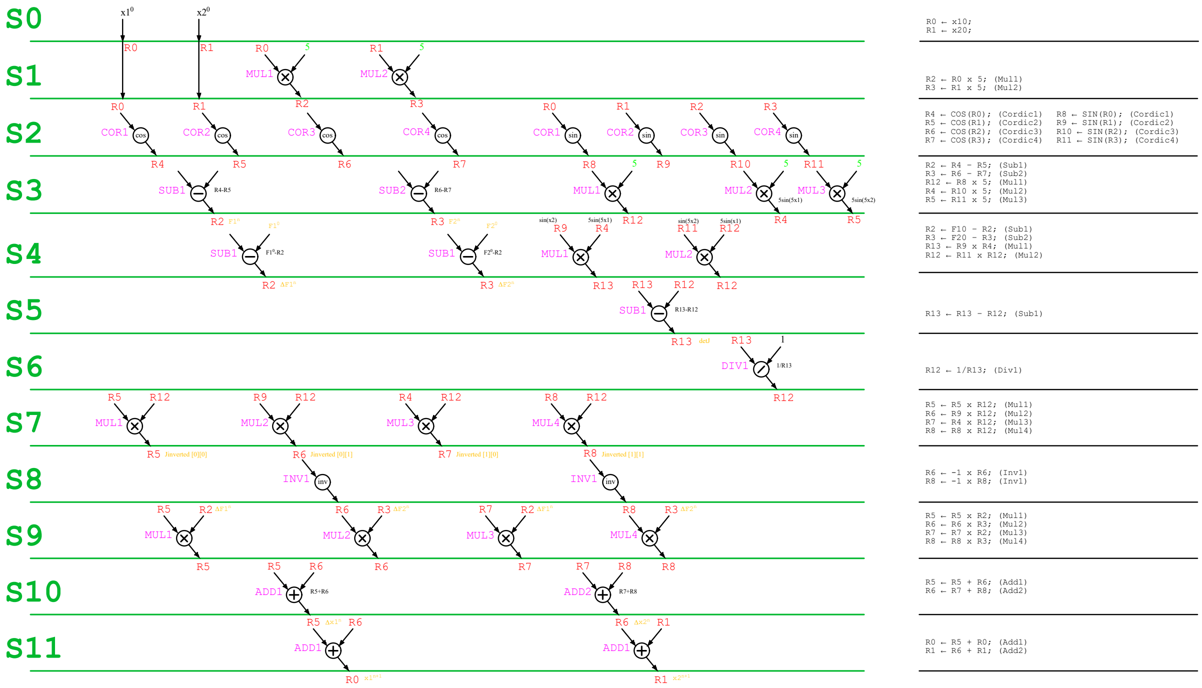


Figure 6 - 4 Allocation and Timing diagram for the Data Path part of Selective Harmonic Elimination (SHE) module.

6.4.4 Data Path Unit

As can be observed from the Figure 6 - 5 the Data Path unit for solving the transcendetal equations is more complex than previously presented units. Obviously the design could be further simplified, i.e., reduce the number of registers and calculation units. This simplification would result in a trade of speed for less complexity. The less complex the design, the less FPGA resources, i.e., LUTs, is needed for the realization of the design. This paper mainly focuses on speed and clarity, so the design consists of thirteen data registers, four CORDIC units, four multiplication units, two adders, two subtractors, one division unit and one invertor unit, which is implemeted directly in the registers logic.

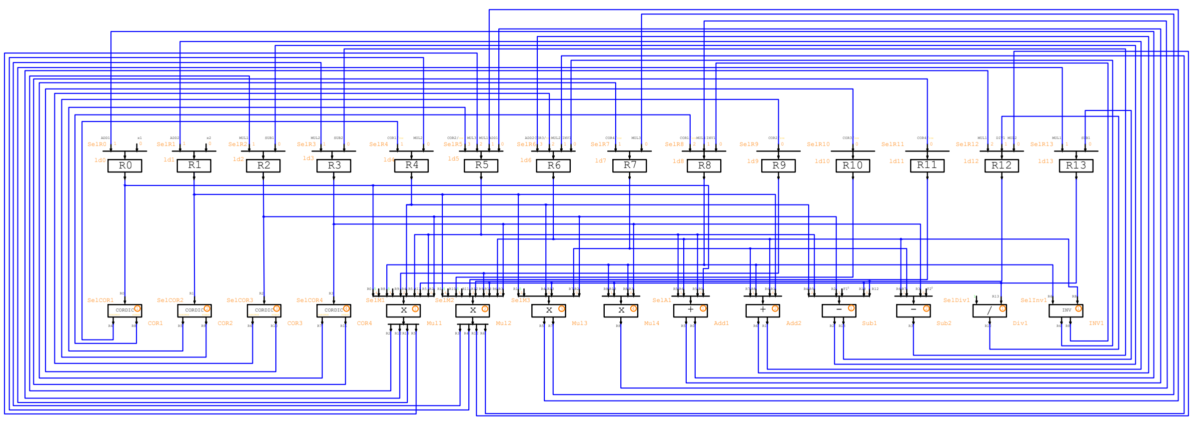


Figure 6 - 5 Register transfer level (RTL) scheme of the Selective Harmonic Elimination Data Path.

6.4.5 Control Unit

Control unit signal specification can be observed in the Table 6 - 1. If the unit design was less complex, i.e., with smaller amount of registers, the control signal length would be smaller, but the number of states would be higher.

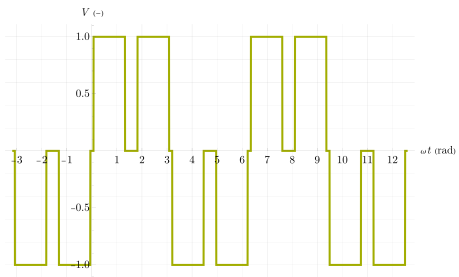
Table 6 - 1 Control signal encoding table for instructions to be processed by the Selective Harmonic Elimination (SHE) alogrithm Module.

State	RTL Code	51 ld0	50 ld1	49 ld2	48 ld3	47 ld4	46 ld5	45 ld6	44 ld7	43 ld8	42 ld9	41 ld10	40 ld11	39 ld12	38 ld13	37 SelR0	36 SelR1	35 SelR2	34 SelR3	33 SelR4	32 SelR5[1]	31 SelR5[0]	30 SelR6[1]	29 SelR6[0]	28 SelR7	27 SelR8[1]	26 SelR8[0]	25 SelR9	24 SelR10
S0	R0 ← x10; R1 ← x20;	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S1	R1 ← R0 x 5; (Mul1) R3 ← R1 x 5; (Mul2)	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
S2	R4 ← COS(R0); (Cordic1) R5 ← COS(R1); (Cordic2) R6 ← COS(R2); (Cordic3) R7 ← COS(R3); (Cordic4) R8 ← SIN(R0); (Cordic1) R9 ← SIN(R1); (Cordic2) R10 ← SIN(R2); (Cordic3) R11 ← SIN(R3); (Cordic4)	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	1	1	1	1	0	1	1	0	0	0
S3	R2 ← R4 – R5; (Sub1) R3 ← R6 – R7; (Sub2) R12 ← R8 x 5; (Mul1) R4 ← R10 x 5; (Mul2) R5 ← R11 x 5; (Mul3)	0	0	1	1	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
S4	R2 ← F10 – R2; (Sub1) R3 ← F20 – R3; (Sub2) R13 ← R9 x R4; (Mul1) R12 ← R11 x R12; (Mul2)	0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S5	R13 ← R13 – R12; (Sub1)	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S6	R12 ← 1/R13; (Div1)	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S7	R5 ← R5 x R12; (Mul1) R6 ← R9 x R12; (Mul2) R7 ← R4 x R12; (Mul3) R8 ← R8 x R12; (Mul4)	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	0	0
S8	R6 ← -1 x R6; (Inv1) R8 ← -1 x R8; (Inv1)	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S9	R5 ← R5 x R2; (Mul1) R6 ← R6 x R3; (Mul2) R7 ← R7 x R2; (Mul3) R8 ← R8 x R3; (Mul4)	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	0	0
S10	R5 ← R5 + R6; (Add1) R6 ← R7 + R8; (Add2)	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
S11	R0 ← R5 + R0; (Add1) R1 ← R6 + R1; (Add2)	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	1	0	0	0	0	0

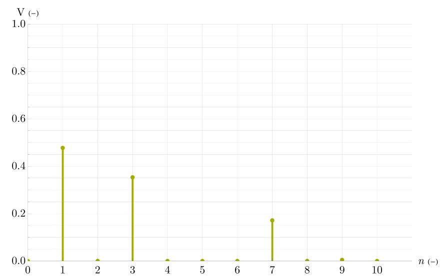
State	23 SelR11	22 SelR12[1]	21 SelR12[0]	20 SelR13	19 SelCor1	18 SelCor2	17 SelCor3	16 SelCor4	15 SelMul1[2]	14 SelMul1[1]	13 SelMul1[0]	12 SelMul2[2]	11 SelMul2[1]	10 SelMul2[0]	9 SelMul3[1]	8 SelMul3[0]	7 SelMul4	6 SelSub1[1]	5 SelSub0[0]	4 SelSub2	3 SelDiv1	2 SelInv1	1 SelAdd1	0 SelAdd2	CS
S0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	52’hC000000000000
S1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	52’h3000C00009000
S2	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	52’hFF03D80F0000
S3	0	1	0	0	0	0	0	0	0	1	1	0	1	1	1	0	0	1	0	1	0	0	0	0	52’h3C08100406E50
S4	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0	0	52’h300C000104820
S5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	52’h4000000000
S6	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	52’h10000400000
S7	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	0	0	0	0	0	0	0	52’h7C00A4002580
S8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	52’h280000000004
S9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	52’h7800A4000000
S10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	52’h600060000003
S11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	52’hC003060000000

6.4.6 Inverter output voltage analysis for Verilog implementation

The simulated VSI output phase voltage, when the SHE algorithm is employed for the modulation index $m = 1$, is depicted in the Figure 6 - 6a. The harmonic analysis for the presented waveform can be observed in the Figure 6 - 6b. As mentioned in previous sections, please note that in the 3-phase symmetrical system the triplen harmonics would be eliminated as well. It is evident that the unwanted 5th harmonic has been successfully eliminated. The calculated angles after ten iterations of the NR algorithm are $\alpha_1 = 0.06341$ rad and $\alpha_2 = 1.320098$ rad.

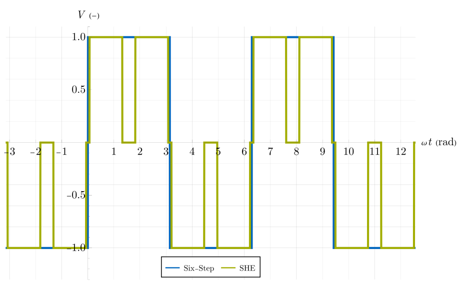


(a) Waveform output of a two level Voltage Source Inverter when the Selective Harmonic Elimination method is applied with Verilog calculated angles. The Voltage value is normalized to a DC link voltage.

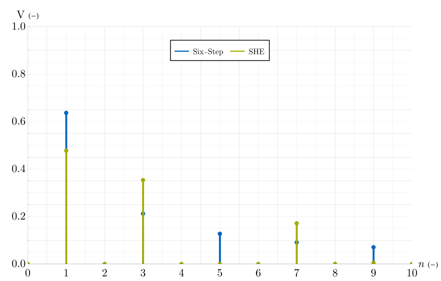


(b) Waveform harmonics analysis of a output voltage of a two level Voltage Source Inverter utilising switching angles for the Selective Harmonic Elimination calculated in Verilog unit. The Voltage value is normalized to a DC link voltage.

Figure 6 - 6 Selective Harmonic Elimination voltage waveform and harmonic analysis.



(a) Comparison of a Waveform output of a two level Voltage Source Inverter when the Selective Harmonic Elimination method is or is not applied with Verilog calculated angles. The Voltage value is normalized to a DC link voltage.



(b) Comparison of a Waveform harmonics analysis of a output voltage of a two level Voltage Source Inverter with and without the Selective Harmonic Elimination. The Voltage value is normalized to a DC link voltage.

Figure 6 - 7 Comparison of a Six-Step and Selective Harmonic Elimination voltage waveforms and harmonic analyses.

6.5 Simulation results

The detail of the ending part of simulation with result of SHE alogrithm after the 10th NR iteration can be seen in Figure 6 - 8. The whole simulation run is depicted in the Figure 6 - 9.

The clock signal frequency in simulation was set to 25 MHz to emulate low cost FPGA capabilities.

Figure 6 - 8 The ending part of the Verilog simulation the Selective Harmonic Elimination (SHE) algorithm. The result are in registers R0 and R1.

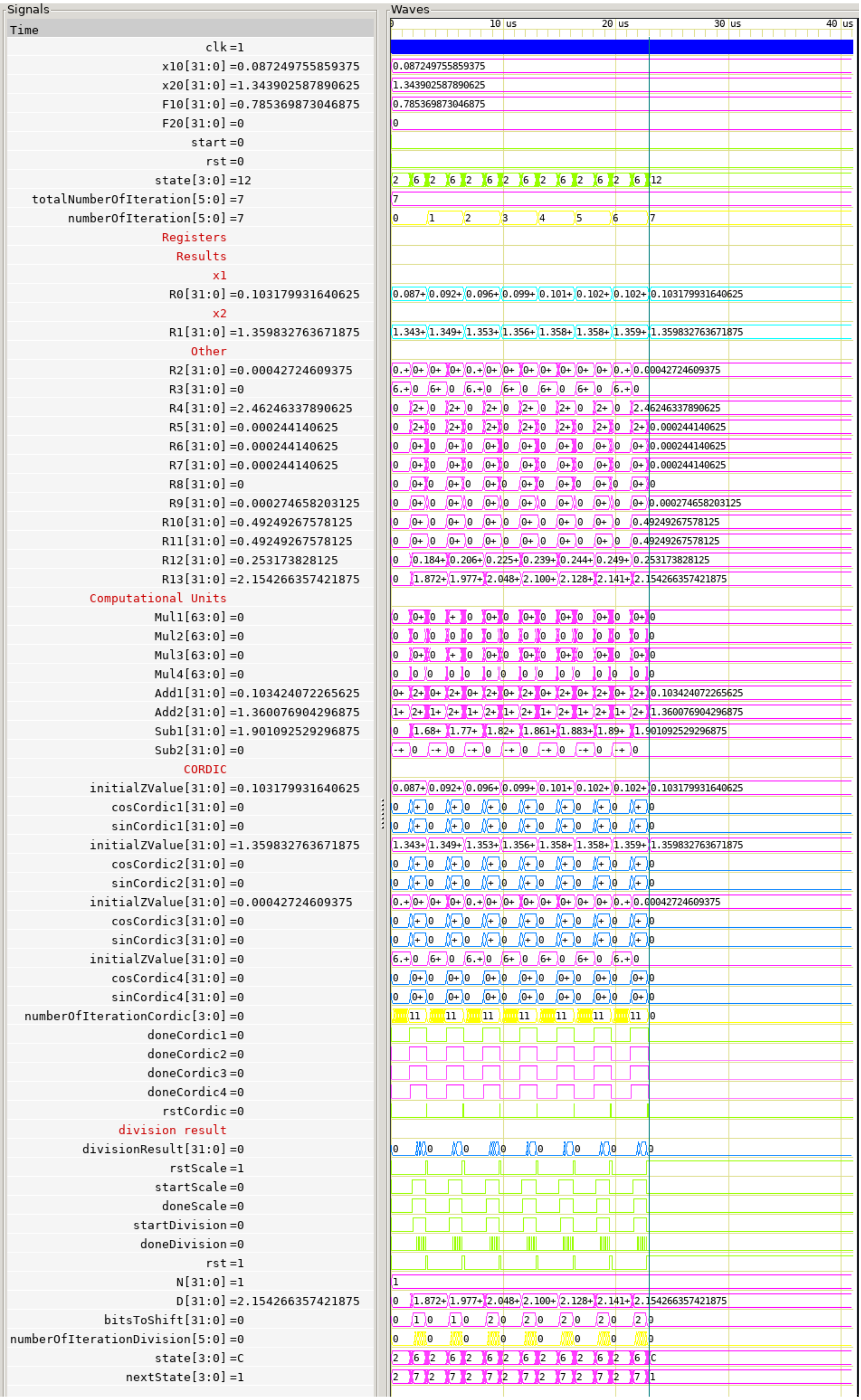


Figure 6 - 9 The complete Verilog simulation of Selective Harmonic Elimination (SHE) algorithm. The result are in registers R0 and R1.

Conclusion

This paper introduces FPGA module designed for solving the SHE algorithm in near real-time. The module comprises two additional submodules, both discussed in this paper. These submodules include units for calculating the division of two arbitrary values and a CORDIC unit suitable for calculating $sine$ and $cosine$ functions.

The primary objective of this paper was to design speed-optimized modules capable of near real-time calculations. The outcomes of this paper can serve as a starting point for future research in designing FPGA modules for controlling electric drives or in creating the Hardware-in-Loop Systems.

References

[1] LTD, Potential Ventures; INC, SolarFlare Communications. Cocotb. In: *Cocotb website* [online]. [B.r.] [visited on 2023-10-08]. Available from: https://www.cocotb.org/.

[2] SNYDER, Wilson. Verilator. In: *Verilator website* [online]. [B.r.] [visited on 2023-10-08]. Available from: https://www.veripool.org/verilator/.

[3] ADVANCED MICRO DEVICES, Inc. Divider Generator LogiCORE™ IP. In: *Intellectual Property* [online]. [B.r.] [visited on 2023-10-01]. Available from: https://www.xilinx.com/products/intellectual-property/divider.html.

[4] BURKE, Tom. Verilog Fixed point math library. In: *GitHub* [online]. [B.r.] [visited on 2023-10-01]. Available from: https://github.com/freecores/verilog_fixed_point_math_library.

[5] MEYER-BÄSE, Uwe. *Digital signal processing with field programmable gate arrays*. 4th ed. Berlin: Springer, 2014. ISBN 978-3-642-45308-3.

[6] VOLDER, Jack E. The CORDIC Trigonometric Computing Technique. *IRE Transactions on Electronic Computers*. 1959, roč. EC-8, č. 3, pp. 330–334. Available from DOI: 10.1109/TEC.1959.5222693.

[7] WALTHER, J. S. A Unified Algorithm for Elementary Functions. In: *Proceedings of the May 18-20, 1971, Spring Joint Computer Conference*. Atlantic City, New Jersey: Association for Computing Machinery, 1971, pp. 379–385. AFIPS '71 (Spring). ISBN 9781450379076. Available from DOI: 10.1145/1478786.1478840.

[8] PATEL, Hasmukh S.; HOFT, Richard G. Generalized Techniques of Harmonic Elimination and Voltage Control in Thyristor Inverters: Part I--Harmonic Elimination. *IEEE Transactions on Industry Applications*. 05/1973, roč. IA-9, č. 3, pp. 310–317. ISSN 0093-9994. Available from DOI: 10.1109/TIA.1973.349908.

[9] PATEL, Hasmukh S.; HOFT, Richard G. Generalized Techniques of Harmonic Elimination and Voltage Control in Thyristor Inverters: Part II --- Voltage Control Techniques. *IEEE Transactions on Industry Applications*. 09/1974, roč. IA-10, č. 5, pp. 666–673. ISSN 0093-9994. Available from DOI: 10.1109/TIA.1974.349239.

[10] MÜLLNER, F.; NEUDORFER, H.; SCHMIDT, E. Modelling and precalculation of additional losses of inverter fed asynchronous induction machines for traction applications. In: *International Aegean Conference on Electrical Machines and Power Electronics and Electromotion, Joint Conference*. 2011, pp. 415–420. Available from DOI: 10.1109/ACEMP.2011.6490634.

[11] TAGHIZADEH, H.; TARAFDAR HAGH, M. Harmonic elimination of multilevel inverters using particle swarm optimization. In: *2008 IEEE International Symposium on Industrial Electronics*. Cambridge, UK: IEEE, 06/2008, pp. 393–396. ISBN 978-1-4244-1665-3. Available from DOI: 10.1109/ISIE.2008.4677093.

[12] ORTIZ-ESPINOZA, Alexandro; MENDEZ-FLORES, Efrain; PONCE-CRUZ, Pedro; MACIAS-HIDALGO, Israel; MOLINA-GUTIERREZ, Arturo. PWM with Selective Harmonic Elimination Using Optimization Inspired on Earthquakes for AC Electric Drives. In: *2020 5th International Conference on Control and Robotics Engineering (ICCRE)*. Osaka, Japan: IEEE, 04/2020, pp. 135–139. ISBN 978-1-72816-791-6. Available from DOI: 10.1109/ICCRE49379.2020.9096462.

[13] ABDELQAWEE, I. M.; ABDEL-RAHIM, Naser M. B.; MANSOUR, Hajji. SELECTIVE HARMONIC ELIMINATION PWM VOLTAGE SOURCE INVERTER BASED ON GENETIC ALGORITHM. 2015. Available also from: https://api.semanticscholar.org/CorpusID:38827373.

[14] WANG, Chenxu; ZHANG, Qi; YU, Wensheng; YANG, Kehu. A Comprehensive Review of Solving Selective Harmonic Elimination Problem with Algebraic Algorithms. *IEEE Transactions on Power Electronics*. 2023, pp. 1–20. ISSN 0885-8993, ISSN 1941-0107. Available from DOI: 10.1109/TPEL.2023.3327280.

[15] CHIASSON, J.N.; TOLBERT, L.M.; MCKENZIE, K.J.; DU, Z. A Complete Solution to the Harmonic Elimination Problem. *IEEE Transactions on Power Electronics*. 03/2004, roč. 19, č. 2, pp. 491–499. ISSN 0885-8993. Available from DOI: 10.1109/TPEL.2003.823207.

[16] BALOW, Writwik; HALDER, T. A Selective Harmonic Elimination (SHE) Technique for the Multi-Leveled Inverters. In: *2018 IEEE Electron Devices Kolkata Conference (EDKCON)*. Kolkata, India: IEEE, 11/2018, pp. 625–630. ISBN 978-1-5386-6415-5. Available from DOI: 10.1109/EDKCON.2018.8770415.

[17] BURENEVA, Olga I.; KAIDANOVICH, Olga U. FPGA-based Hardware Implementation of Fixed-point Division using Newton-Raphson Method. In: *2023 IV International Conference on Neural Networks and Neurotechnologies (NeuroNT)*. 2023, pp. 45–47. Available from DOI: 10.1109/NeuroNT58640.2023.10175844.

Appendix A: List of and Abbreviations

A.1 List of abbreviations

CORDIC	Coordinate Rotation Digital Computer
CPU	Central Processing Unit
DC	Direct Current
FOSS	Free and Open-Source Software
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
IP	Intellectual property
ISA	Instruction Set Architecture
LUT	Look Up Table
NR	Newton Raphson
RTL	Register Transfer Level
SHE	Selective Harmonic Elimination
VCD	Value Change Dump
VSI	Voltage Source Inverter