



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

1	Introduction	1
2	Notes on all of the circuit designs in Verilog	2
3	Calculating the division of fixed point numbers.....	3
3.1	Newton Rapshon algorithm for calculating the division	3
3.2	IP Block Design	4
3.2.1	Top module design	4
3.2.2	Allocation and Timing	5
3.2.3	Data Path Module	6
3.2.4	Control Unit	7
3.3	Calculating number of bits to shift the denominator.....	8
3.4	Simulation results	8
4	Using CORDIC to calculate trigonometric functions	13
4.1	Theory	13
4.1.1	Example of calculation	15
4.2	Python Implementation	15
4.3	IP Block Design	18
4.3.1	Top module design	18
4.3.2	Allocation and Timing	19
4.3.3	Data Path Module	20
4.3.4	Control Unit	23
4.4	Simulation results	23
5	Simple set of nonlinear equations solved by a Newton-Raphson algorithm using custom circuit implementation	27
5.1	Theory	27
5.2	IP Block Design	28
5.2.1	Top module design	28
5.2.2	Allocation and Timing	29
5.2.3	Data Path Unit	30
5.2.4	Control Unit	32
5.3	Simulation results	32
6	Selective Harmonic Elimination	34
6.1	Theory	34
6.2	Simplification for Verilog and High level implementation	36
6.3	High level implementation	37
6.4	IP Block Design	39
6.4.1	Algorithm Block Diagram	39
6.4.2	Top module design	40

6.4.3	Allocation and Timing	41
6.4.4	Data Path Unit	43
6.4.5	Control Unit	43
6.4.6	Inverter output voltage analysis for Verilog implementation	44
	Conclusion	45
	References	47
Appendix A	List of and Abbreviations	48
A.1	List of abbreviations.....	48

LIST OF FIGURES

3 - 1	Top module design for the division unit module block design.	5
3 - 2	Allocation and timing diagram for the Data Path Unit part of the division module.	6
3 - 3	Register Transfer Level (RTL) scheme of the Data Path Unit part of the division module.	7
3 - 4	Selected signals of simulation of division $N/D = 10 / 7$. The correct result in $R0$ is obtained after two iterations (reg numberOfIterations).	10
3 - 5	Selected signals of simulation of division $N/D = 1 / 0.25$. The correct result in $R0$ is obtained after five iterations (reg numberOfIterations).	10
3 - 6	Selected signals of simulation of division $N/D = 1 / (-0.25)$. The correct result in $R0$ is obtained after five iterations (reg numberOfIterations).	11
3 - 7	Selected signals of simulation of division $N/D = 304.03215 / (-0.25)$. The correct result in $R0$ is obtained after five iterations (reg numberOfIterations).	11
3 - 8	Selected signals of simulation of division $N/D = 10 / (519)$. The correct result in $R0$ is obtained after two iterations (reg numberOfIterations).	12
4 - 1	Top module design for the CORDIC module block design.	19
4 - 2	Allocation and timing diagram for the Data Path Unit part of the CORDIC IP.	20
4 - 3	Register transfer level RTL scheme of the CORDIC IP Data Path Unit IP.	21
4 - 4	The whole Verilog simulation of CORDIC algorithm for determining the sinus and cosinus values of angle $\theta = -1.2479$ rad. The value of sinus and cosinus based on the current iteration is also calculated in this algorithm approach. The result is passed to the registers R9 and R10.	24
4 - 5	The detail of the last iteration of the Verilog simulation of CORDIC algorithm for determining the sinus and cosinus values of angle $\theta = -1.2479$ rad. The result is passed to the registers R9 and R10.	25
4 - 6	The whole Verilog simulation of CORDIC algorithm for determining the sinus and cosinus values of angle $\theta = 10.7195129$ rad. The value of sinus and cosinus based on the current iteration is also calculated in this algorithm approach. The result is passed to the registers R9 and R10.	25
4 - 7	The whole Verilog simulation of CORDIC algorithm for determining the sinus and cosinus 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 the registers R9 and R10.	26
5 - 1	Top module design for the simple Newton-Raphson (NR) calculation unit module block design.	29
5 - 2	Allocation and timing diagram for the Data Path Unit part of the simple (NR) module. ...	30
5 - 3	Register Transfer Level (RTL) scheme of the Data Path Unit part of the simple Newton-Raphson (NR) calculation IP.	31
5 - 4	The whole Verilog simulation of a simple Newton-Raphson (NR) algorithm. The result is may be seen in registers R1 and R2 after the fifth iteration of the algorithm.	33
6 - 1	34
6 - 2	Block Diagram of the Selective Harmonic Elimination (SHE) using Newton-Raphson algorithm.	40

6 - 3	Top module design for the Selective Harmonic Elimination unit (SHE).	41
6 - 4	Allocation and Timing diagram for the Data Path Unit part of Selective Harmonic Elimination (SHE) module.	42
6 - 5	Register transfer level (RTL) scheme of the Selective Harmonic Elimination Data Path Unit.	43
6 - 6	44
6 - 7	44

LIST OF TABLES

3 - 1 Control signal encoding table for instructions to be processed by the Division Module. ... 7

4 - 1 Control signal encoding table for instructions to be processed by the CORDIC Module... 23

5 - 1 Control signal encoding table for instructions to be processed by the simple Newton-
Raphson (NR) alogrithm solve Module. 32

6 - 1 Control signal encoding table for instructions to be processed by the Selective Harmonic
Elimination (SHE) alogrithm solve Module..... 44

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 Intellectual 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 synthesized 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 Notes on all of the circuit designs in Verilog

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 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 a specific application. Academics and numerous companies opt for open-source and open-hardware approaches to prevent vendor lock-in. Once the design and algorithm are thoroughly understood, they can be initially 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) have been 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.

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 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*.

$$x_0 = \frac{48}{17} - \frac{32}{17}D, \quad (3 - 1)$$

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

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'sb000000000000000010_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 the 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.

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

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

$$\frac{dF(x_i)}{dx} = F'(x_i) = \frac{F(x_{i+1}) - F(x_i)}{x_{i+1} - x_i}. \quad (3 - 3)$$

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 eq. 3 - 3 and derivating the function $F(x_i)$ the obtained algorithm for a value x_{i+1} is obtained from eq. 3 - 4.

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

Usually, the iterative algorithm is stopped, when the value $F(x_{i+1}) - F(x_i)$ (called 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 N-R 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 is 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 is 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 design.

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 whole 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.

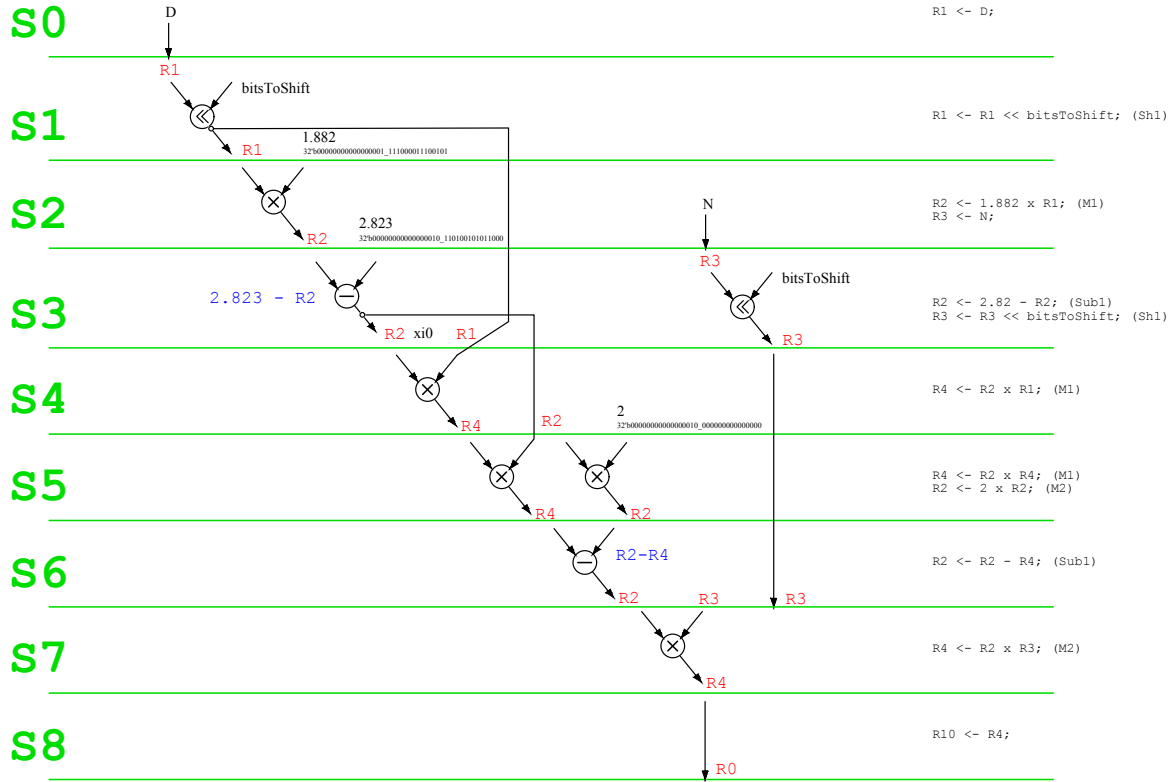


Figure 3 - 2 Allocation and timing diagram for the Data Path Unit part of the division module.

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.

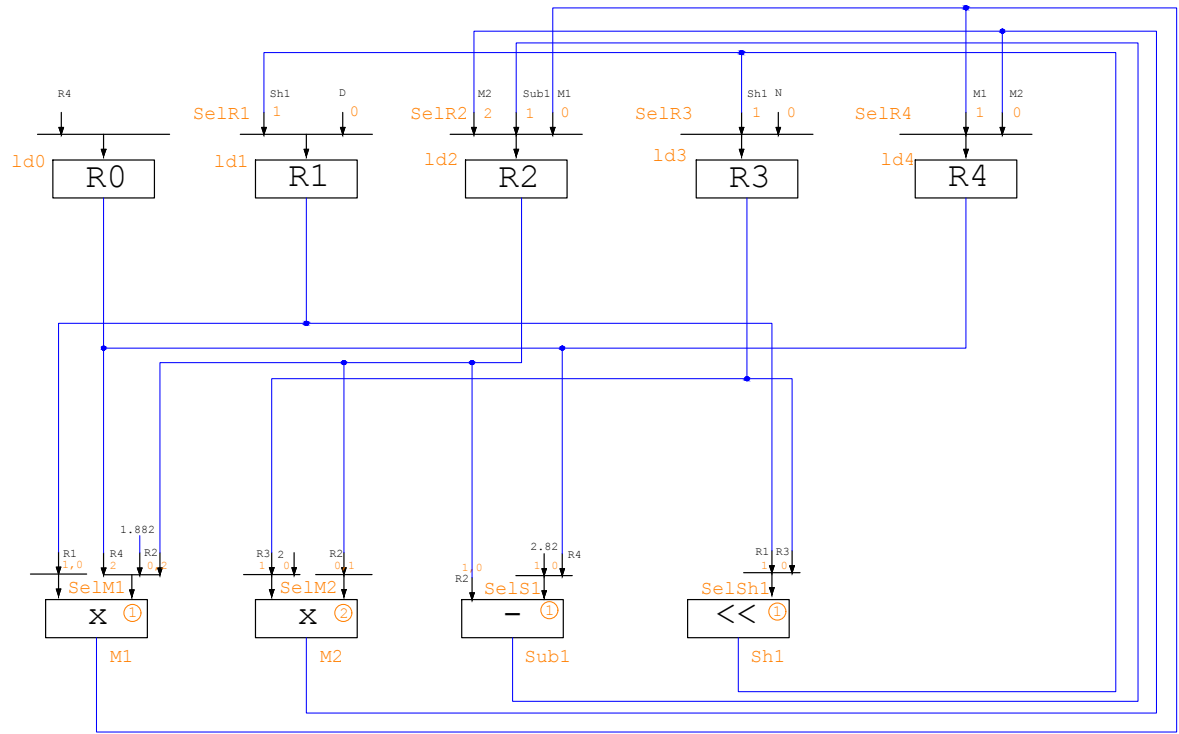


Figure 3 - 3 Register Transfer Level (RTL) scheme of the Data Path Unit part of the division module.

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 passed to 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	$R1 \leftarrow D;$	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	2000h
S1	$R1 \leftarrow R1 \ll 32; (Sh1)$	0	1	0	0	0	1	0	0	0	0	1	0	0	0	0	15'h2210
S2	$R2 \leftarrow 1.882 \times R1; (M1)$ $R3 \leftarrow N;$	0	0	1	1	0	0	0	0	0	0	0	0	1	0	0	15'h1804
S3	$R2 \leftarrow 2.82 - R2; (Sub1)$ $R3 \leftarrow R3 \ll 32; (Sh1)$	0	0	1	1	0	0	0	1	1	0	0	0	0	0	0	15'h18C0
S4	$R4 \leftarrow R2 \times R1; (M1)$	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	420h
S5	$R4 \leftarrow R2 \times R4; (M1)$ $R2 \leftarrow 2 \times R2; (M2)$	0	0	1	0	1	0	1	0	0	1	0	1	0	0	0	15'h1528
S6	$R2 \leftarrow R2 - R4; (S1)$	0	0	1	0	0	0	0	1	0	0	0	0	0	0	1	15'h1081
S7	$R4 \leftarrow R2 \times R3; (M2)$	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	15'h402
S8	$R0 \leftarrow R4;$	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.

```
1 at every negative edge of clock or positive edge of reset
2   if(rst)
3       scaleToShift = 0;
4       scaleToShiftInternal = 1;
5       started = 0;
6   end if
7   else if (start)
8       started = 1;
9   end else if
10
11 at every positive edge of clock
12   if (compare <= 32'b000000000000000001_0000000000000000)
13       done = 1;
14       started = 0;
15       scaleToShift = scaleToShiftInternal;
16   end if
17   else
18       done = 0;
19       scaleToShiftInternal = scaleToShiftInternal + 1;
20   end if
21
22 at every positive edge of clock
23   if(start)
24       compare <= DInternal >> scaleToShiftInternal;
25   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.

VCD simulation output waveforms are depicted on the following Figures. The simulations were conducted for arbitrarily selected values of N and 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* when the *start* signal is set on low.

```

1  timescale 1ns/1ns
2  #10; // wait for 10 units of time
3  #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
4  N = 32'b000000000100110000_0000100000000000; D=32'
   b1111111111111111_1100000000000000; // set the numerator to N =
   304.03125, denominator to D = -0.25
5  #10 rstScale = 0; // wait for 10 units of time and stop the reset of
   scaling unit
6  #10 startScale = 1; // start the algorithm for scaling unit
7  #20 rst = 1; start = 0; // reset the division unit
8  #30 rst = 0; // stop resetting of the division unit
9  #20 start = 1; // start the division unit
10 #20 start = 0;
11 #1000; // wait 1000 units of time
12 $finish; // finish the simulation

```

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

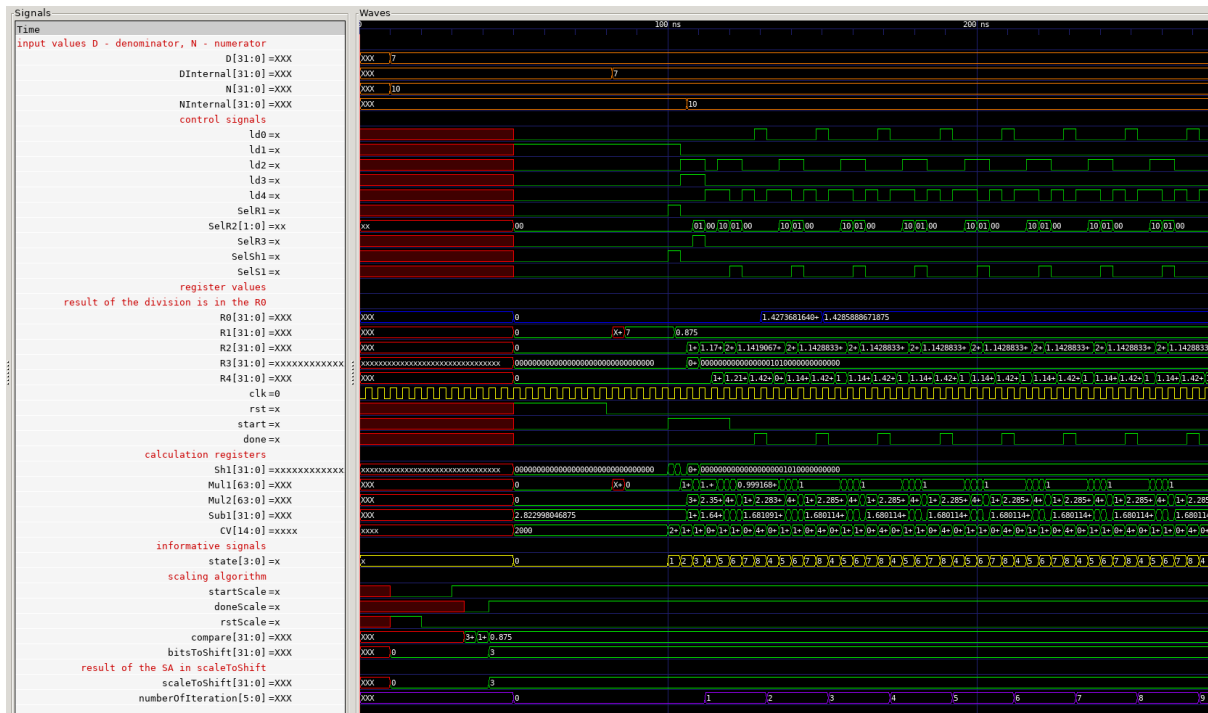


Figure 3 - 4 Selected signals from simulation of division $N/D = 10 / 7$. The correct result in R0 is obtained after two iterations (reg numberOfIterations).

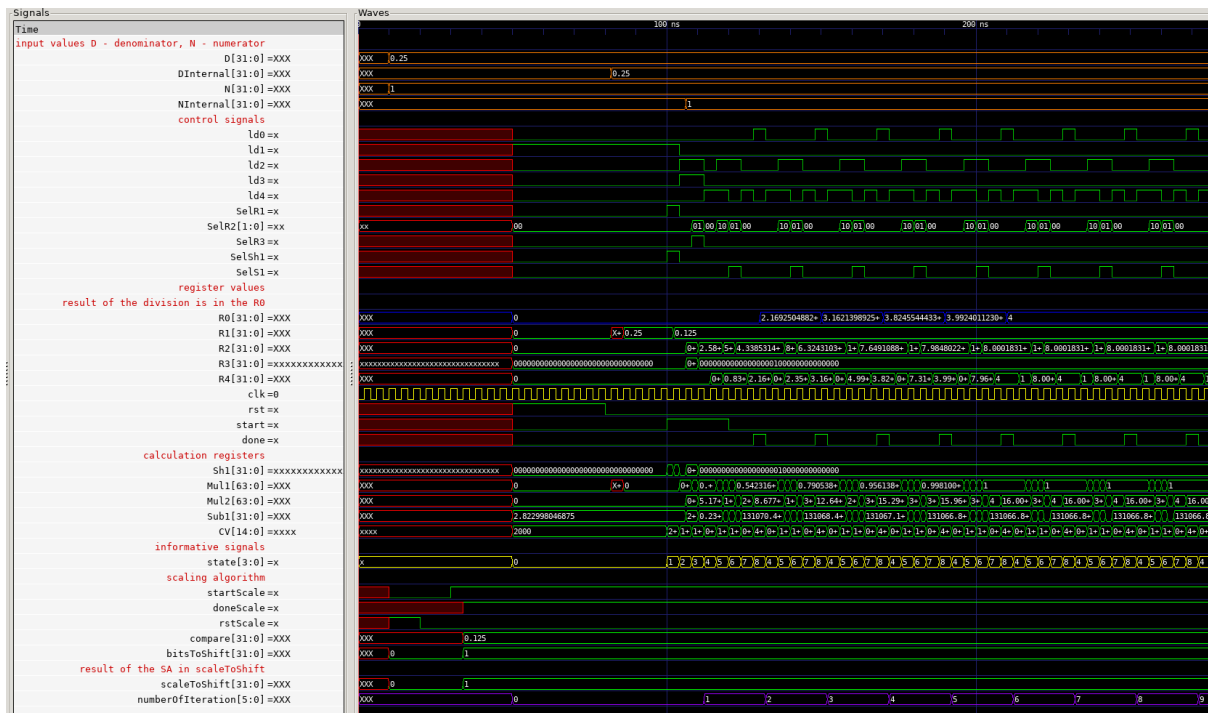


Figure 3 - 5 Selected signals from simulation of division $N/D = 1 / 0.25$. The correct result in R0 is obtained after five iterations (reg numberOfIterations).

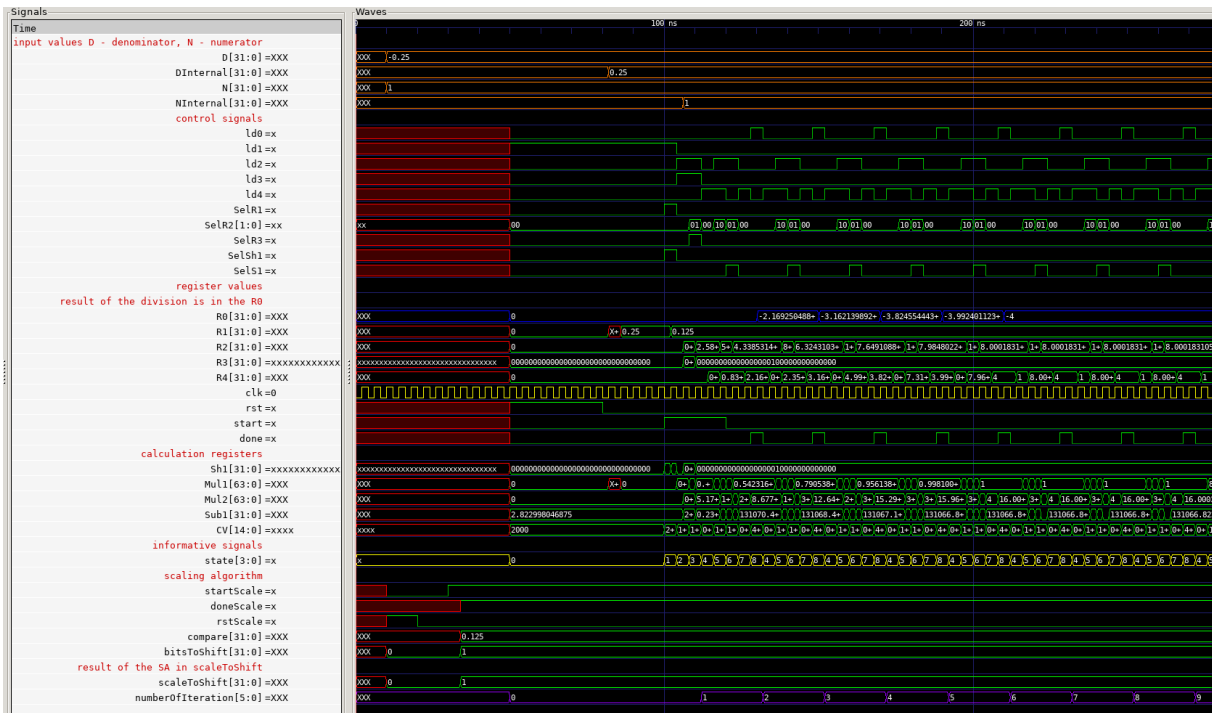


Figure 3 - 6 Selected signals from simulation of division $N/D = 1 / (-0.25)$. The correct result in R0 is obtained after five iterations (reg numberOfIterations).

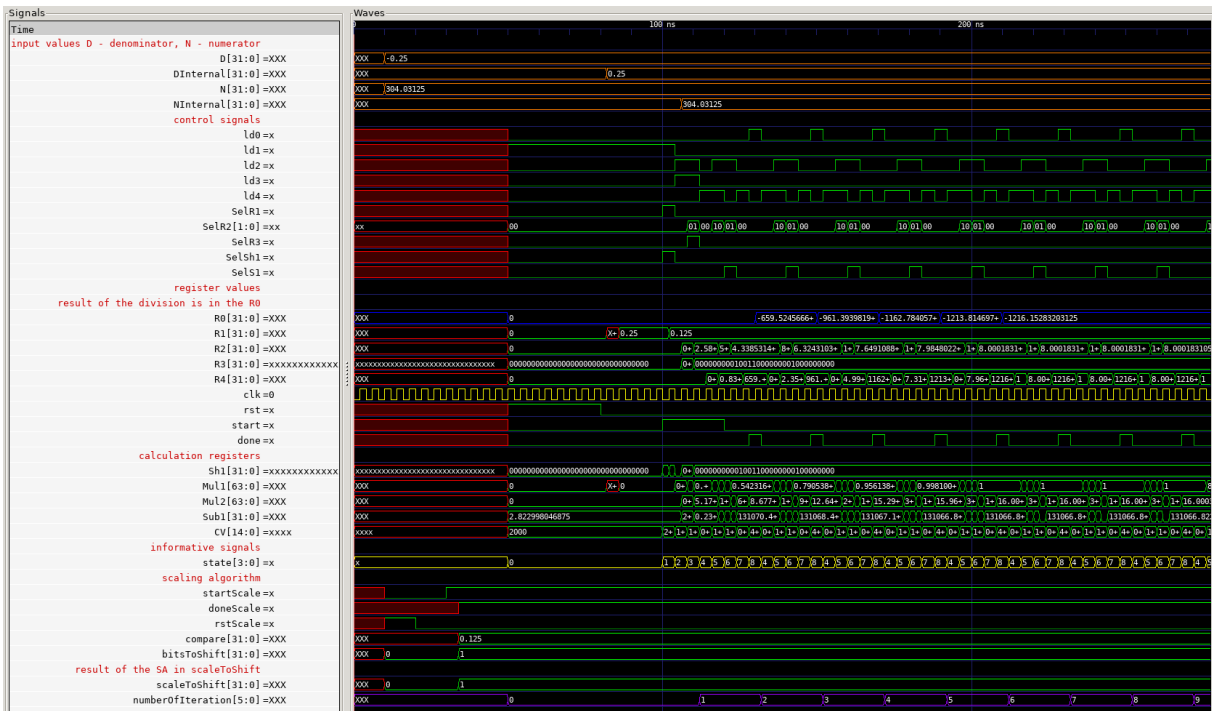


Figure 3 - 7 Selected signals from simulation of division $N/D = 304.03215 / (-0.25)$. The correct result in R0 is obtained after five iterations (reg numberOfIterations).

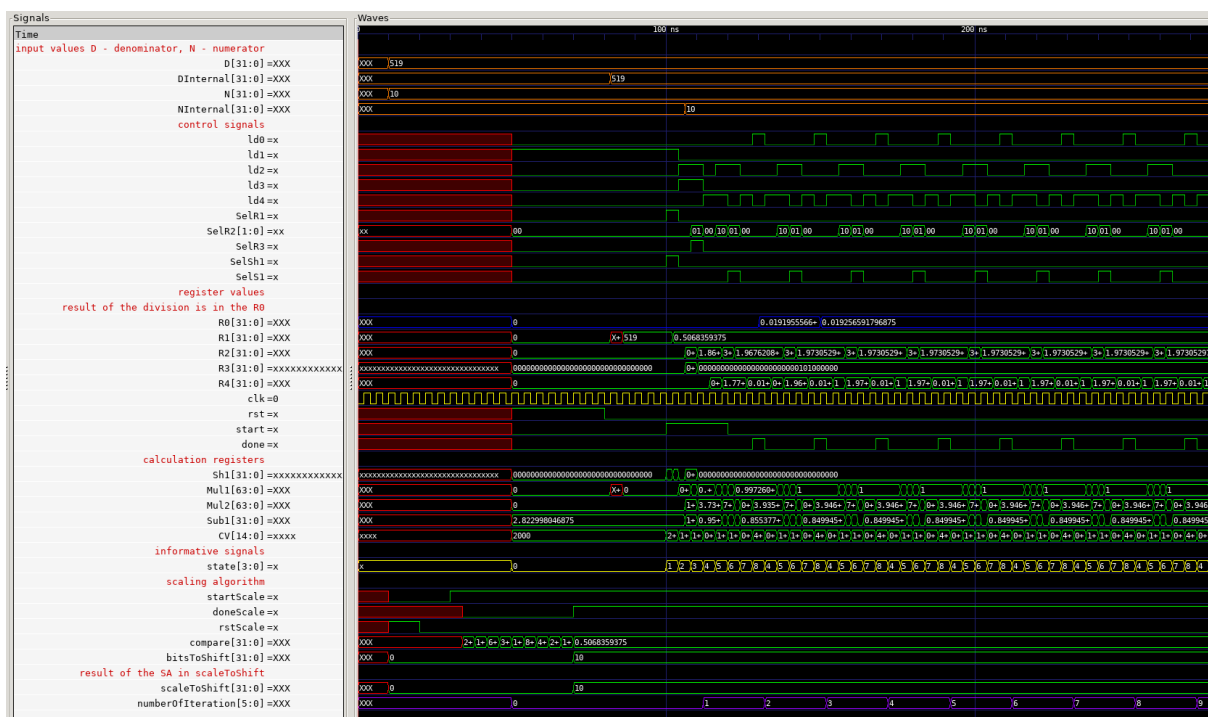


Figure 3 - 8 Selected signals from simulation of division $N/D = 10 / (519)$. The correct result in R0 is obtained after two iterations (reg numberOfIterations).

4 Using CORDIC to calculate trigonometric functions

There are numerous ways how to calculate the trigonometric functions. To gain more flexibility the Coordinate Rotation Digital Computer (CORDIC) was chosen above the Look-Up Table (LUT) implementation.

The LUT method may be fast, but the accuracy depends on the size of the table. When using the CORDIC the precision depends on number of performed iterations of the algorithm. The modified algorithm may be used to calculate non-trivial functions, such as hyperbolic functions, square roots, multiplications, divisions, exponentials and logarithms. [5] In this work only the calculation of *sinus* and *cosinus* functions is used.

4.1 Theory

The theory of the first CORDIC was proposed by Volder in [6]. This algorithm computes a coordinate conversion between rectangular (x, y) and polar (R, θ) coordinates. The algorithm was then generalized by Walther in [7] to include circular, linear and hyperbolic transforms. This paper utilizes only circular transforms to calculate *sinus* and *cosinus* functions. Only the most basic approach of the algorithm will be presented.

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

$$\begin{pmatrix} x_R \\ y_R \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} x_{in} \\ y_{in} \end{pmatrix}, \quad (4 - 1)$$

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} was rotated.

Then when simplifying the equation

$$\begin{pmatrix} x_R \\ y_R \end{pmatrix} = \cos(\theta) \begin{pmatrix} 1 & -\tan(\theta) \\ \tan(\theta) & 1 \end{pmatrix} \begin{pmatrix} x_{in} \\ y_{in} \end{pmatrix} \quad (4 - 2)$$

it can be seen, that only multiplication by scaling factor of precalculated values of $\cos(\theta)$, multiplication by $\tan(\theta)$, subtraction and addition operations are needed. However, the multiplication by $\tan(\theta)$ can be interchanged. The interchange may be done for angles θ for which the equation 4 - 3 is true. The when implementing the algorithm to the FPGA the multiplication may be swapped for signed right bit shift.

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

When the values $x_{in} = 1$ and $y_{in} = 0$ are used, the result for *sinus* and *cosinus* may be easily obtained from x_R and y_R as expressed in the equation 4 - 4.

$$\begin{aligned} x_R &= x_{in} \cos(\theta) - y_{in} \sin(\theta) = |\theta = 0| = \cos(\theta) \\ y_R &= x_{in} \sin(\theta) + y_{in} \cos(\theta) = |\theta = 0| = \sin(\theta) \end{aligned} \quad (4 - 4)$$

The algorithm may be further simplified by expecting that the algorithm is designed to use more than 6 iterations and thus the scaling constant represented by multiplying *cosinus* of different θ values converges to 0,60725. So there is no need to precalculate all the scaling values only the convergent value may be used. In this paper the precalculated values are passed from the custom LUT module to the

main algorithm.

As can be seen from the section *Example of calculation* section or the algorithm theory itself, it needs to be determined, if the angle for which the vector is rotated in the next iteration should be in a positive direction (counter-clockwise) or negative direction (clockwise). For that, the set of the equations is expanded and new value z_i added. The complete set of equations which are used in the implementation are as follows.

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

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

$$\sigma_{i+1} = \begin{cases} -1, & \text{if } z_{i+1} < 0 \\ 1, & \text{if } z_{i+1} > 0 \\ 0, & \text{if } z_{i+1} = 0 \end{cases} \quad (4-6)$$

The algorithm as presented calculates the correct values for *sinus* and *cosinus* functions only in the first and fourth quadrant ($3\pi/2$ to $\pi/2$ counter-clockwise). For usage in the whole 2π range, corresponding actions before the 0. iteration must be made.

The algorithm must make checks, to determine the quadrant, where the desired angle θ for which the *sinus* and *cosinus* functions are to be calculated. This is done by *if* statements at the algorithm values initialization and at the final function value calculation. If the desired argument of the functions is not in the first or fourth quadrant then the angle is transferred from the actual quadrant to the first or fourth quadrant. Based on the quadrant, to which the angle is transformed, the σ_i value is set. The corresponding *if* statements at the algorithm initialization are presented in the pseudocode 4 - 1.

Similar *if* statements are used at the final calculation of *sinus* and *cosinus* values. The *if* statements are presented in the pseudocode 4 - 2.

The pseudocodes use *initialZValue* as a desired angle θ , for which to calculate the function values, *zValue* as a temporary value for calculating the iterations for z_i variables, *sigmaValue* for temporary value holding 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.

```

1 if((initialZValue > 1.5707)&(initialZValue < 3.141592))
2     sigmaValue = -1
3     zValue = initialZValue - 3.141592
4 else if((initialZValue > 3.141592)&(initialZValue < 4.7123))
5     sigmaValue = 1
6     zValue = initialZValue - 3.141592
7 else
8     zValue = initialZValue
9     sigmaValue = 1
10 end

```

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

```

1 if((initialZValue > 1.5707)&(initialZValue < 3.141592))

```

```

2   resultCos = - resultCos
3   resultSin = resultSin
4 else if((initialZValue > 3.141592)&(initialZValue < 4.7123))
5   resultCos = - resultCos
6   resultSin = - resultSin
7 end

```

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

4.1.1 Example of calculation

The general approach of CORDIC algorithm may be explained on the example for calculating the *sinus* and *cosinus* values for the angle $\theta = 57, 535^\circ$. Firstly, the angle may be deconstructed in the base angles, for which the equation 4 - 3 is true. In this example the is deconstructed as $57, 535 = 45 + 25, 565 - 14, 03$.

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

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

$$1. \text{ iteration } \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \cos(25, 565^\circ) \begin{pmatrix} 1 & -2^{-1} \\ 2^{-1} & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}, \quad (4 - 8)$$

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

Then after substitution the value of x_2 and y_2 may be obtained.

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

From the equation 4 - 10 the values x_2 and y_2 represent the value of $\cos(57, 535^\circ)$ and $\sin(57, 535^\circ)$ respectively.

4.2 Python Implementation

The CORDIC algorithm was for simplicity prototyped in python. This turned out very beneficial as the debugging of the code is much faster. The less complex and abstract python code may help with understanding and creating the designed algorithms more than Mathematica which uses some higher abstraction layers to make calculations optimized and easier for more complex problems. But when designing the low level mathematical algorithms, the lower and easier language the more easy is then to implement the design in Verilog or any other hardware description language.

The python code was as well used to precalculate the LUT for scaling factor and arcus tangens values for z_i calculations.

For the clarity, the python implementation is presented in the code 4 - 3. The code also calculates the error of the CORDIC calculated value from the python math library functions.

```

1 import math

```

```

2
3 # Defining starting values and empty arrays
4 totalNumberOfIterations = 12 # 12 - best tradeoff between value and
  iterations
5 atanValues = []
6 scalingValues = [1]
7 initialXValueCordic = 1
8 initialYValueCordic = 0
9 # initialZValueCordic = 1.248 # angle for which to calculate cordic
10 # initialZValueCordic = - 1.248 # angle for which to calculate cordic
11 # initialZValueCordic = - 6.7194 # angle for which to calculate cordic
12 initialZValueCordic = 10.7194824 # angle for which to calculate cordic
13 initialSigmaValueCordic = 1
14
15 for x in range(totalNumberOfIterations):
16     # Generating arcus tangens values of precalculated angles based on
  number of iterations
17     atanValues.append(math.atan(1*2**(-x)))
18     # Generating precalculated scaling values based on a number of
  iterations
19     scalingValues.append(scalingValues[x]*math.cos(atanValues[x]))
20
21 print("atanValues: ", atanValues)
22 print("scalingValues: ", scalingValues)
23
24 print("*-+-+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+*")
25 print("\n")
26 print("initialZValue original: ", initialZValueCordic)
27
28 # Moving angle to interval [0,2Pi]
29 if initialZValueCordic > 0:
30     while initialZValueCordic > (2*3.141592):
31         initialZValueCordic = initialZValueCordic - 2*3.141592
32 else:
33     while initialZValueCordic < (-2*3.141592):
34         initialZValueCordic = initialZValueCordic + 2*3.141592
35
36
37 print("initialZValue after moving to [0,2Pi] interval: ",
  initialZValueCordic)
38 print("\n")
39 print("*-+-+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+*")
40
41 # Checking the initial value and moving it in the interval
42 if (initialZValueCordic > 1.5707) and (initialZValueCordic < 3.141592):
43     zValue = initialZValueCordic - 3.141592
44     sigmaValue = -1
45     print("value in second q")

```



```

46 elif (initialZValueCordic > 3.141592) and (initialZValueCordic < 4.7123):
47     zValue = initialZValueCordic - 3.141592
48     sigmaValue = 1
49     print("value in third q")
50 elif (initialZValueCordic < 0):
51     sigmaValue = -1
52     zValue = initialZValueCordic
53     print("value in fourth q")
54 else:
55     zValue = initialZValueCordic # For angle
56     sigmaValue = initialSigmaValueCordic # For +- next angle
57     print("value in first")
58
59 # Passing starting values to the calculation values
60 xValue = initialXValueCordic # For cos
61 yValue = initialYValueCordic # For sin
62
63
64 # CORDIC ALGORITHM
65 for x in range(totalNumberOfIterations):
66
67     # Calculating next values of the current iteration x
68     xNextValue = xValue - (sigmaValue*yValue)*2**(-x)
69     yNextValue = yValue + (sigmaValue*xValue)*2**(-x)
70     zNextValue = zValue - sigmaValue * atanValues[x]
71
72     # Determining the signum of next angle (addition or subtraction)
73     if zNextValue >= 0:
74         sigmaNextValue = 1
75     else:
76         sigmaNextValue = -1
77
78     # Values for new iteration
79     xValue = xNextValue
80     yValue = yNextValue
81     zValue = zNextValue
82     sigmaValue = sigmaNextValue
83
84     print("iteration:", x, "xValue:", xValue, "yValue:", yValue, "zValue:",
85           zValue, "sigmaValue:", sigmaValue, "\n")
86
87 # Calculating results by scaling the result values from CORDIC by the
88   scalingValue which depends on number of iterations which were made
89 resultCos = scalingValues[x-1] * xValue
90 resultSin = scalingValues[x-1] * yValue
91
92 # Changing results sign based on the rotation of the initialZValueCordic
93 if (initialZValueCordic > 1.5707) and (initialZValueCordic < 3.141592):

```

```

92     resultCos = - resultCos
93 elif (initialZValueCordic > 3.141592) and (initialZValueCordic < 4.7123):
94     resultCos = - resultCos
95     resultSin = - resultSin
96
97 # Calculating values based on the math library
98 mathResultCos = math.cos(initialZValueCordic)
99 mathResultSin = math.sin(initialZValueCordic)
100
101 # Calculating the error of CORDIC calculated values from the python math
    functions
102 errorCos = abs(resultCos) - abs(mathResultCos)
103 errorSin = abs(resultSin) - abs(mathResultSin)
104
105 # Results printing
106 print("*-+-+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+*")
107 print("CORDIC results:")
108 print("cos: ", resultCos)
109 print("sin: ", resultSin)
110 print("scaleFactor: ", scalingValues[totalNumberOfIterations-1])
111 print("\n")
112 print("MATH results:")
113 print("cos: ", mathResultCos)
114 print("sin: ", mathResultSin)
115 print("\n")
116 print("error CORDIC-MATH:")
117 print("cos: ", errorCos)
118 print("sin: ", errorSin)

```

Code 4 - 3 Python code of CORDIC implementation.

After the python implementation and debugging has been finalized, the circuit Verilog implementation of the algorithm could be initiated. Same as for the Division Unit IP, presented in *Calculating the division of fixed point numbers* section, the Data Path, Control Unit and Top Module was designed. This approach based on the application specific circuit design should be by its nature faster and more safe than creating the custom CPU with reduced and customized ISA.

4.3 IP Block Design

4.3.1 Top module design

The top module design of the CORDIC IP is shown in the picture 4 - 1. As can be seen, the structure is very much similar to the Division Unit top module. When using the approach to create a customized circuit for algorithm the flow of creating the top modules is likely to be similar with minor differences in signals, inputs and variables.

The Data Path Moule in the top design incorporates the precalculated LUTs for *atanValues* and *scalingValues*. The LUT memory module's structure is very simple and therefore the Verilog interpretation is depicted only for *atanValues* variable. The value of *totalNumberOfIterations* is set to be 12 in this implementation, thus the LUT is 12x32 bits in size. Obviously the already presented custom fixed point

$Q32.15$ format is required.

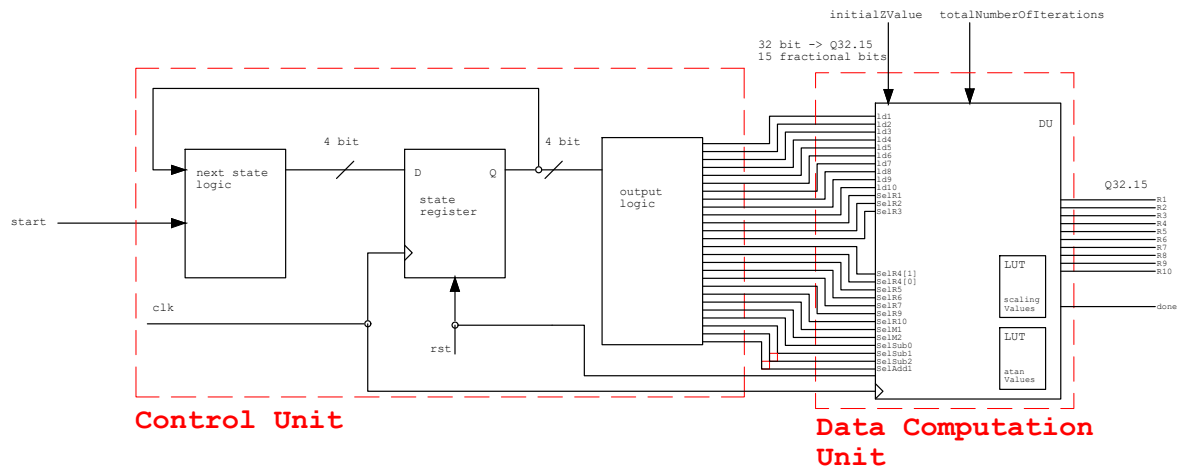


Figure 4 - 1 Top module design for the CORDIC module block design.

4.3.2 Allocation and Timing

In the picture 4 - 2 the allocation and timing diagram is depicted. As can be seen, the if statements which are implemented in the control unit are documented here as well. The explanation why the if statements are needed is stated in the *CORDIC Theory* section. As stated in the section for *CORDIC Control Unit* there are two approaches of iteration cycles. The designer may choose jump from $S4$ to $S2$ for faster algorithm or from $S6$ to $S2$ for demonstrative approach. The jumps in the allocation and timing diagram are not shown.

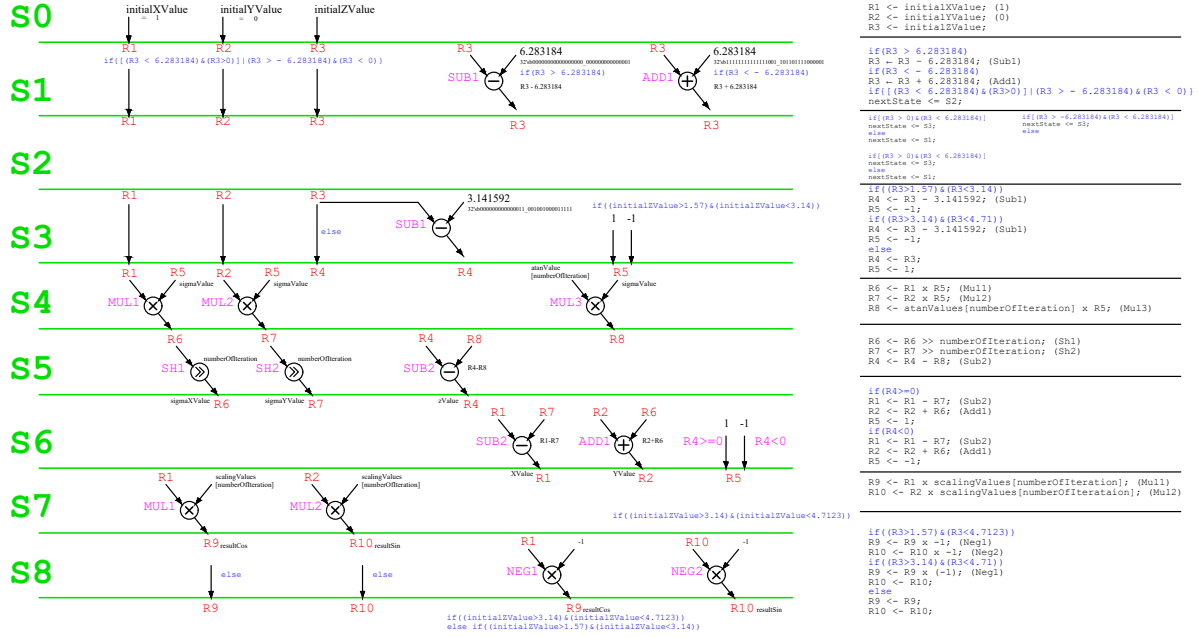


Figure 4 - 2 Allocation and timing diagram for the Data Path Unit part of the CORDIC IP.

4.3.3 Data Path Module

The picture 4 - 3 visualize the Data Path part of the Top Module design including calculation and storing units. The memory LUTs for *atanValues* and *scalingValues* are not depicted as a separate registers but as inputs to the calculation units. The results of *sinus* and *cosinus* functions, in python implementation named as *resultSin* and *resultCos* are saved to registers R9 and R10. The **NEG** blocks aren't in fact implemented as a standalone blocks for making negative numbers. The negation is activated in a corresponding target register when the appropriate **SeIR_x** is activated. (where *x* is here the number of a corresponding register R9 or R10)

As was stated before, 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.

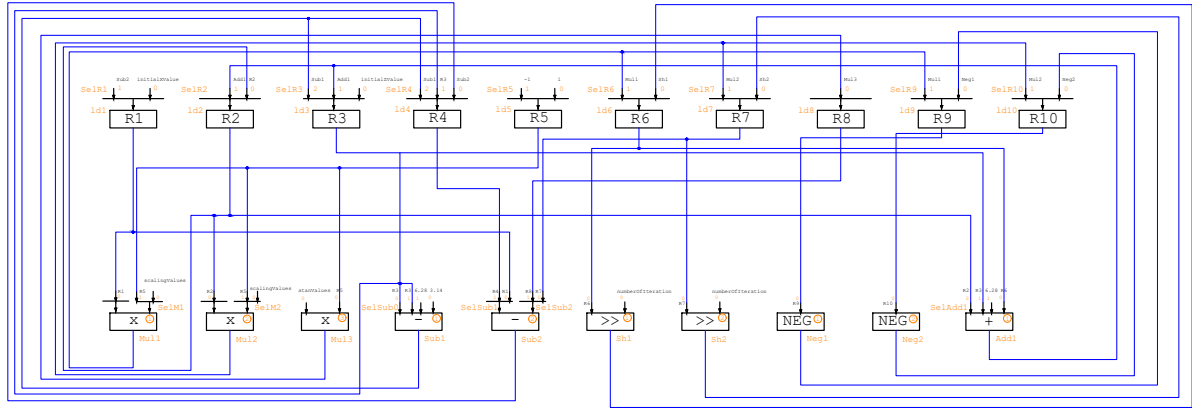


Figure 4 - 3 Register transfer level RTL scheme of the CORDIC IP Data Path Unit IP.

```

1 module atanValuesCordicLUT(index, returnValue);
2
3 input [3:0] index;
4 output reg signed [31:0] returnValue;
5
6
7 always@(index)
8 begin
9     case(index)
10         4'b0000: returnValue = 32'sb00000000000000000000_110010010000111; //
11             0.7853981633974483
12         4'b0001: returnValue = 32'sb00000000000000000000_011101101011000; //
13             0.4636476090008061
14         4'b0010: returnValue = 32'sb00000000000000000000_001111101011011; //
15             0.24497866312686414
16         4'b0011: returnValue = 32'sb00000000000000000000_000111111101010; //
17             0.12435499454676144
18         4'b0100: returnValue = 32'sb00000000000000000000_000011111111101; //
19             0.06241880999595735
20         4'b0101: returnValue = 32'sb00000000000000000000_000001111111111; //
21             0.031239833430268277
22         4'b0110: returnValue = 32'sb00000000000000000000_000000111111111; //
23             0.015623728620476831
24     endcase
25 end

```

```

17      4'b0111: returnValue = 32'sb000000000000000000_0000000111111111; //
0.007812341060101111
18      4'b1000: returnValue = 32'sb000000000000000000_0000000111111111; //
0.007812341060101111
19      4'b1001: returnValue = 32'sb000000000000000000_0000000011111111; //
0.0019531225164788188
20      4'b1010: returnValue = 32'sb000000000000000000_0000000001111111; //
0.0009765621895593195
21      4'b1011: returnValue = 32'sb000000000000000000_0000000000111111; //
0.0004882812111948983
22      default: returnValue = 32'sb000000000000000000_0000000000000000; // 0
23  endcase
24 end
25 endmodule

```

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

```

1 module scalingValuesCordicLUT(index, returnValue);
2
3 input [3:0] index;
4 output reg signed [31:0] returnValue;
5
6 always@(index)
7 begin
8     case(index)
9         4'b0000: returnValue <= 32'sb000000000000000001_0000000000000000; //
1          1
10         4'b0001: returnValue <= 32'sb000000000000000000_101101010000010; //
0.7071067811865476
11         4'b0010: returnValue <= 32'sb000000000000000000_101000011110100; //
0.6324555320336759
12         4'b0011: returnValue <= 32'sb000000000000000000_100111010001001; //
0.6135719910778964
13         4'b0100: returnValue <= 32'sb000000000000000000_100110111101110; //
0.6088339125177524
14         4'b0101: returnValue <= 32'sb000000000000000000_100110111000111; //
0.6088339125177524
15         4'b0110: returnValue <= 32'sb000000000000000000_100110110111101; //
0.607351770141296
16         4'b0111: returnValue <= 32'sb000000000000000000_100110110111011; //
0.6072776440935261
17         4'b1000: returnValue <= 32'sb000000000000000000_100110110111010; //
0.6072591122988928
18         4'b1001: returnValue <= 32'sb000000000000000000_100110110111010; //
0.6072544793325625
19         4'b1010: returnValue <= 32'sb000000000000000000_100110110111010; //
0.6072533210898753
20         4'b1011: returnValue <= 32'sb000000000000000000_100110110111010; //
0.6072530315291345

```

```

21         default: returnVal <= 32'sb000000000000000000_0000000000000000; //
           0
22     endcase
23 end
24 endmodule

```

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

4.3.4 Control Unit

Same way as in a Division Module Control unit, presented in *Control Unit* section, the control signal encoding table 4 - 1 for Data Path CORDIC unit is created.

The branches of if statements used in the design has been colorcoded in the table for improved clarity. The iteration jumps are not depicted in the control signal table. The jumps may be performed from the step *S4*, when the speed of the calculation is the main concern, or from *S6*, when the algorithm function is presented. The steps *S5* and *S6* are mainly focused on multiplying the result of iteration by the appropriate scaling value and on transforming the results based on the quadrant of the original wanted angle value.

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

State	RTL Code	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	CS
S0	R0 ← totalNumberIterations; R1 ← initialZValue; R2 ← initialYValue; R3 ← initialXValue;	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	27b700000
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	27b100000
S1	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	1	27b100001
S2	if(R3 > 0.460686) → 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	27b0
S2	if(R3 > 0.460686) → 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	
S2	if(R3 > 0.460686) → 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	
S3	if(R3 > 1.570796) → nextState <= S3; CS = 0; R4 ← R1 - R5; (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	27b001400
S3	if(R3 > 1.570796) → nextState <= S3; CS = 0; R4 ← R1 - R5; (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	27b001000
S3	if(R3 > 1.570796) → nextState <= S3; CS = 0; R4 ← R1 - R5; (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	27b000C00
S3	if(R3 > 1.570796) → nextState <= S3; CS = 0; R4 ← R1 - R5; (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	26b000800
S4	R6 ← R1 * R5; (Mul1) R7 ← R2 * R5; (Mul2) R8 ← atanValues[numberOfIteration] * 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	26b300330
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	26b000006
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	26b6418000
S7	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	1	0	0	0	0	0	0	0	0	0	26b6418400
S7	R9 ← R1 * scalingValues[numberOfIteration]; (Mul1) R10 ← R2 * scalingValues[numberOfIteration]; (Mul2)	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	26b600C0
S8	if(R3 > 5.14159265) → nextState <= S3; CS = 0; R9 ← R9 * (-1); (Neg1) R10 ← R10 * (-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	26b60000
S8	if(R3 > 5.14159265) → nextState <= S3; CS = 0; R9 ← R9 * (-1); (Neg1) R10 ← R10 * (-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	26b40000
S8	if(R3 > 5.14159265) → nextState <= S3; CS = 0; R9 ← R9 * (-1); (Neg1) R10 ← R10 * (-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	24b0

4.4 Simulation results

The testbench for testing the design is created with cocotb and simulated with Verilator.

As can be seen when implementing the algorithm where the actual iteration value for *sinus* and *cosinus* is calculated, the number of cycles needed for the final calculation can be calculated

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

where *NoCyc* (-) is the number of cycles and *NoIt* is the number of iterations for the CORDIC algorithm. The 4 value is for *S0-S4* and the multiplication by 5 is because of states *S4-S8*. When the

result of the CORDIC algorithm is calculated only once at the end of the algorithm, the number of iteration can be determined by

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

where the multiplication by value 3 is caused by states *S4–S6*, the addition of 4 is caused by states *S0–S4* and the addition of the 2 is caused by states *S7–S8*.

In the simulation the *numberOfCycles* displayed is more of an index of the cycle, so for angle θ is the number of iterations depicted on Figure 4 - 5 in fact 63 not displayed 62.

The frequency of the clock signal in this design is currently set as 50 MHz.

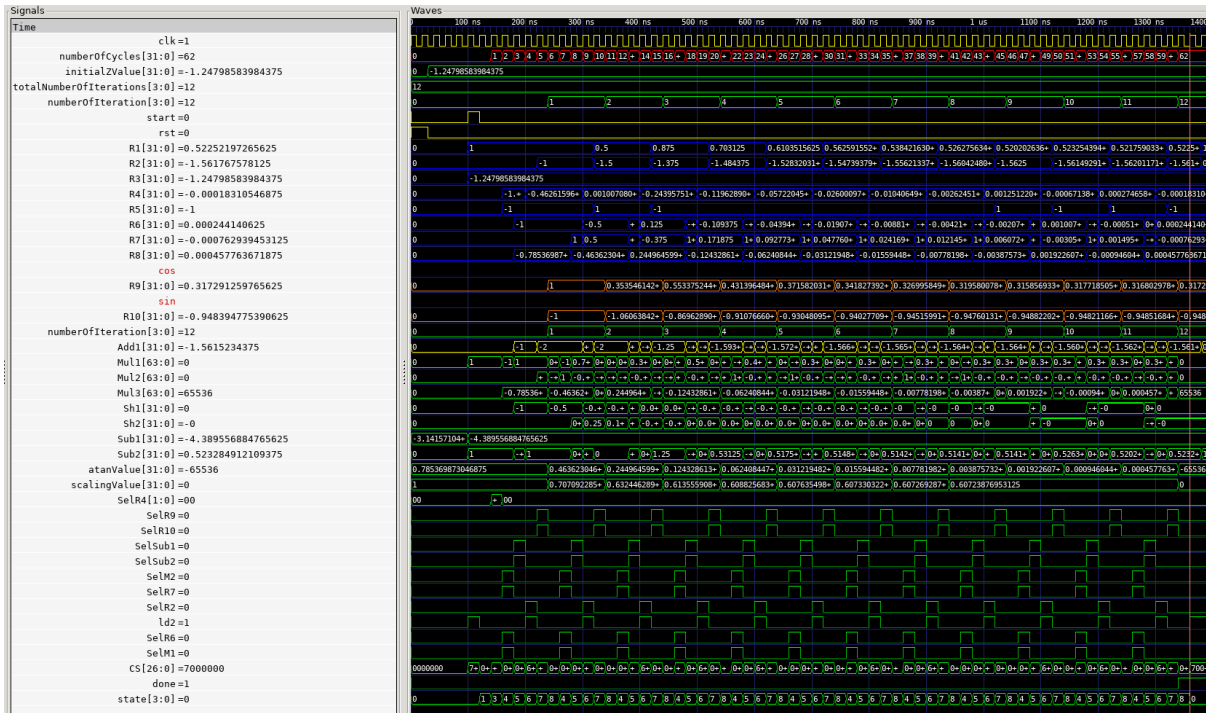


Figure 4 - 4 The whole Verilog simulation of CORDIC algorithm for determining the sinus and cosinus values of angle $\theta = -1.2479$ rad. The value of sinus and cosinus based on the current iteration is also calculated in this algorithm approach. The result is passed to the registers R9 and R10.

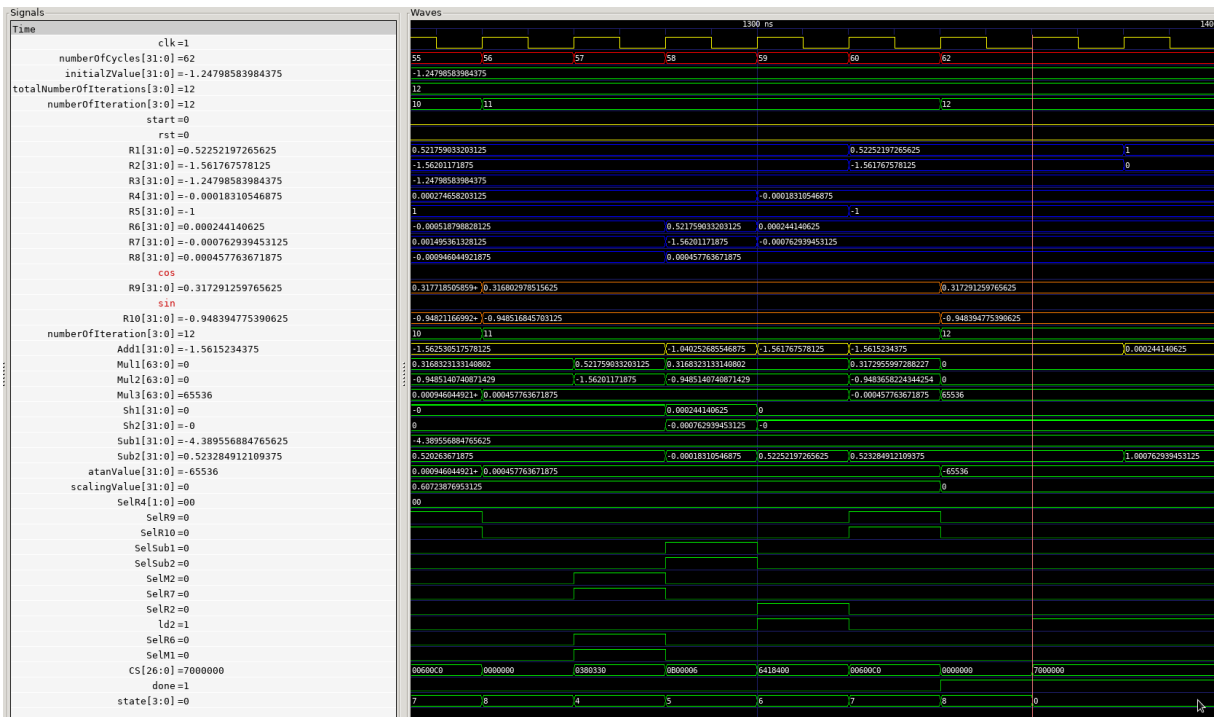


Figure 4 - 5 The detail of the last iteration of the Verilog simulation of CORDIC algorithm for determining the sinus and cosinus values of angle $\theta = -1.2479$ rad. The result is passed to the registers R9 and R10.

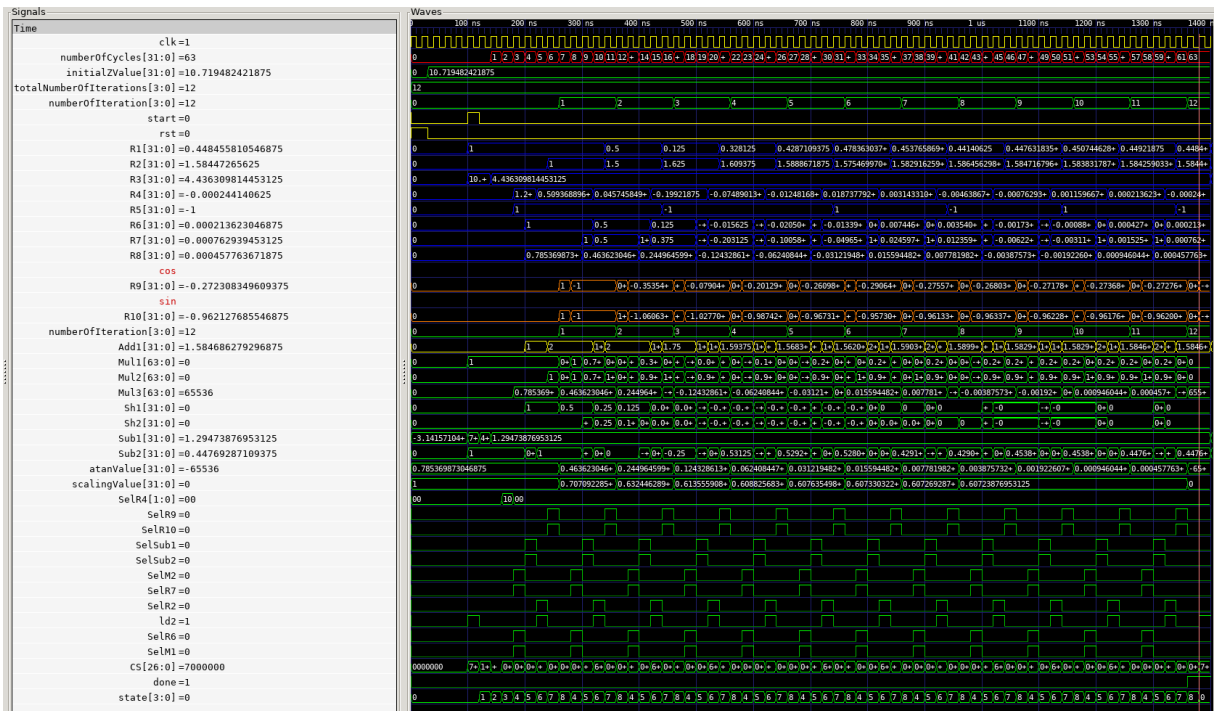


Figure 4 - 6 The whole Verilog simulation of CORDIC algorithm for determining the sinus and cosinus values of angle $\theta = 10.7195129$ rad. The value of sinus and cosinus based on the current iteration is also calculated in this algorithm approach. The result is passed to the registers R9 and R10.

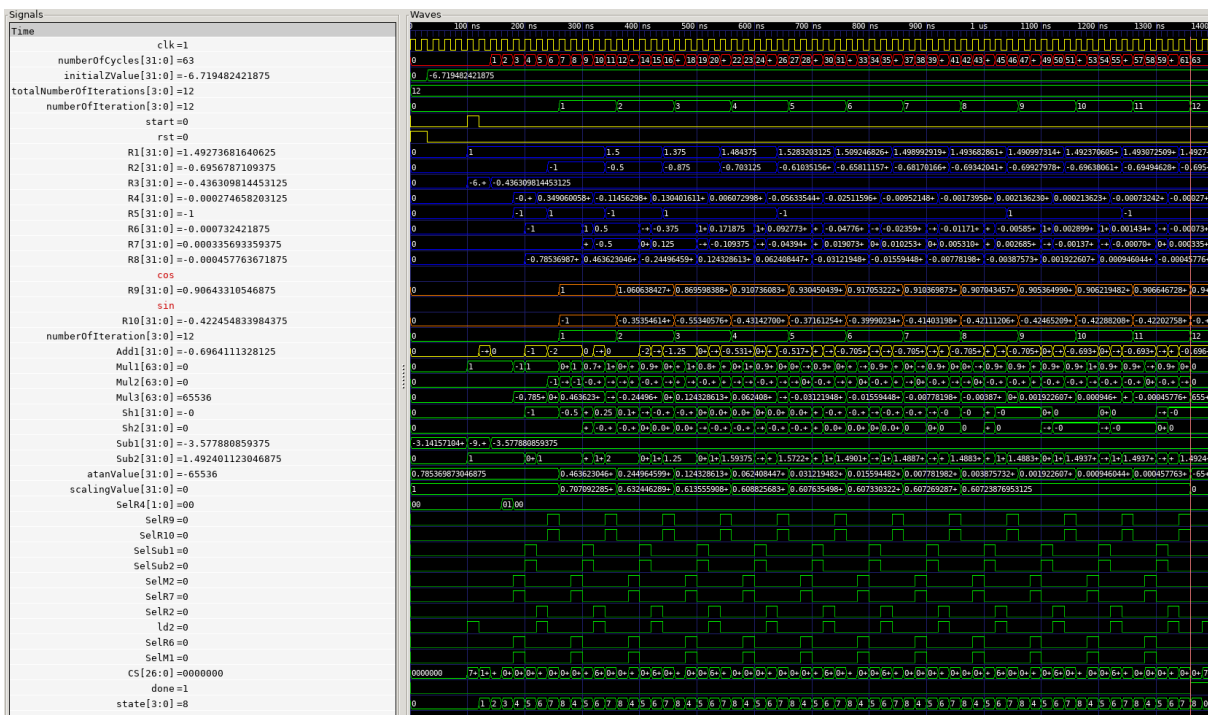


Figure 4 - 7 The whole Verilog simulation of CORDIC algorithm for determining the sinus and cosinus values of angle $\theta = -6.7195129$ rad. The value of sinus and cosinus based on the current iteration is also calculated in this algorithm approach. The result is passed to the registers R9 and R10.

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

All the presented parts in previous sections may be utilized to solve the system of nonlinear equations. This work leads to solving the transcendental equations for Selective Harmonic Elimination. But the best approach is to firstly solve an easier set of equations to determine, if the approach of NR is viable.

5.1 Theory

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

$$F_1(x_1, x_2) = x_1^3 - x_2 - 1, \quad (5 - 1)$$

$$F_2(x_1, x_2) = x_1 - 2x_2 - 2, \quad (5 - 2)$$

where one possible set of solutions x_1 and x_2 yields

$$F_1 = 0, \quad (5 - 3)$$

$$F_2 = 0. \quad (5 - 4)$$

The algorithm could be implemented in a custom CPU with reduced instruction set but for the obvious reasons, eg. speed and complexity of developing own RISC-V, the approach of creating the application specific circuit design was used.

To be able to implement the algorithm to the custom design, the general NR algorithm approach had to be simplified to the most low level implementation. Every single part that could be precalculated was set as a static value at the design step.

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 the start of the algorithm the starting values of x_1^0 and x_2^0 were set as an input to the module. Based on that input the function values at selected starting points were calculated.

As a next step, the so called defect could be calculated using the newly found values of $F_1(x_1^0)$ and $F_2(x_1^0, x_2^0)$

$$\Delta \mathbf{F}^i = \begin{pmatrix} \Delta F_1^i \\ \Delta F_2^i \end{pmatrix} = \begin{pmatrix} F_1^i - F_1^{\text{known solution}} \\ F_2^i - F_2^{\text{known solution}} \end{pmatrix}, \quad (5 - 5)$$

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 \mathbf{J} from vector of functions $(F)(x_1, x_2) = (F_1, F_2)$ is calculated as follows.

$$\mathbf{J}^i = \begin{pmatrix} \frac{dF_1}{dx_1^i} & \frac{dF_1}{dx_2^i} \\ \frac{dF_2}{dx_1^i} & \frac{dF_2}{dx_2^i} \end{pmatrix} = \begin{pmatrix} 3(x_1^i)^2 & -1 \\ 1 & -2 \end{pmatrix}. \quad (5 - 6)$$

As for the general NR algorithm, the inverted value of Jacobian matrix needs to be calculated. The problem is that when using general mathematical software, such as Wolfram Mathematica, the calculation of the inverted value 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, its adjugate 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.

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

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

The adjugate matrix is calculated as follows

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

After the calculation of the reciprocal value of the determinant of the Jakobi matrix and the adjugate matrix, the inverted Jakobi matrix bay be finally calculated

$$\mathbf{J}^{-1i} = \frac{1}{\det(\mathbf{J}^i)} \begin{pmatrix} \text{adj}(\mathbf{J}_{00}^i) & \text{adj}(\mathbf{J}_{01}^i) \\ \text{adj}(\mathbf{J}_{10}^i) & \text{adj}(\mathbf{J}_{11}^i) \end{pmatrix} = \frac{1}{\det(\mathbf{J}^i)} \begin{pmatrix} -2 & -1 \\ 1 & 3(x_1^i)^2 \end{pmatrix}. \quad (5 - 9)$$

Next the $(\Delta x_1^i, \Delta x_2^i)$ is to be calculated by using the inverted Jakobi matrix and the defect.

$$\begin{pmatrix} \Delta x_1^i \\ \Delta x_2^i \end{pmatrix} = \begin{pmatrix} \mathbf{J}_{00}^{-1,i} \Delta F_1^i + \mathbf{J}_{01}^{-1,i} \Delta F_2^i \\ \mathbf{J}_{10}^{-1,i} \Delta F_1^i + \mathbf{J}_{11}^{-1,i} \Delta F_2^i \end{pmatrix}. \quad (5 - 10)$$

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

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

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

5.2 IP Block Design

5.2.1 Top module design

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

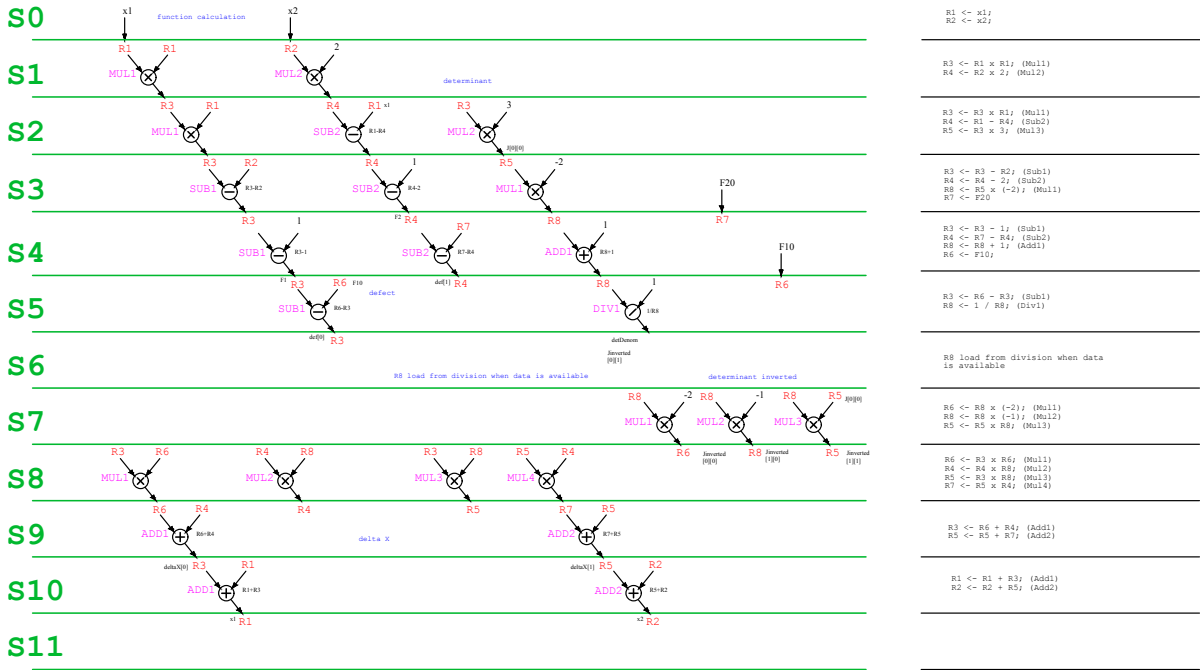


Figure 5 - 2 Allocation and timing diagram for the Data Path Unit part of the simple (NR) module.

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*. When the algorithm has finished the results for x_1 and x_2 are saved in the R1 and R2, the state S11 is set and *done* signal is set to 1. The results then can be driven to another module or unit for further usage. In fact the *done* signal is driven in the Control Unit and can be used in controlling the possible module, where the NR module is only part of the design.

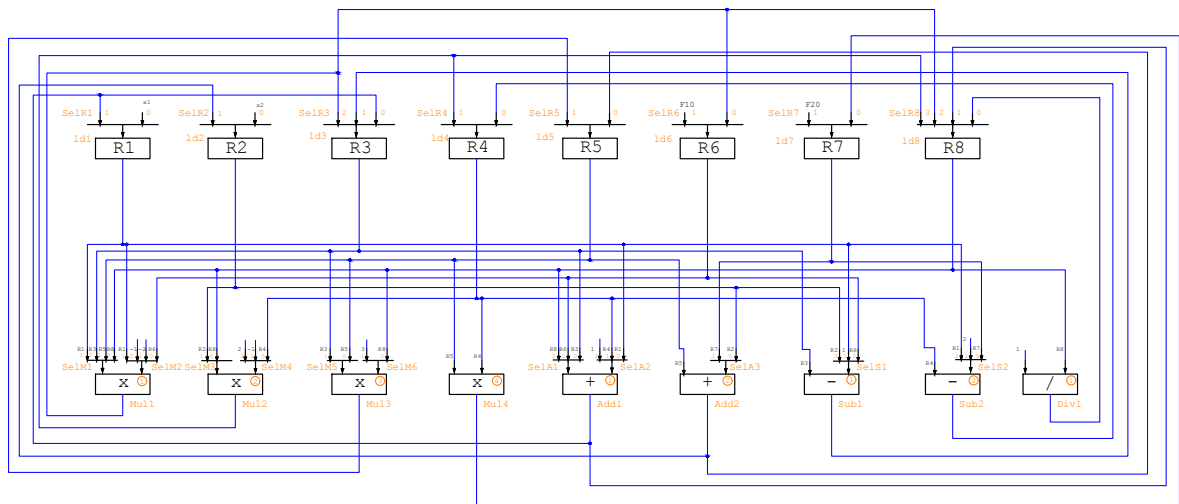


Figure 5 - 3 Register Transfer Level (RTL) scheme of the Data Path Unit part of the simple Newton-Raphson (NR) calculation IP.

5.2.4 Control Unit

The encoding table 5 - 1 shows the steps of the algorithm with a corresponding control signal for the Data Path Unit of the simple NR algorithm Verilog implementation.

The NR algorithm iteration jumps are carried out from the state *S10* to state *S1*, when the numebr of iteration is lower than the set total number of iterations, which is hardcoded to the Control Unit. At this implementation, the total number of iterations is se to be 5. In fact, the end of the NR algorithm should be determined based on the defect value. In this simple example, the value check of the defect is not implemented. The implementation would be simple though. The value of register holding the defect values R3 and R4 would be wired to the control unit in the corresponding steps *S4* and *S5* respectively and the comparison with the desired defect value would be performed. If the defect value was smaller than the desired value, the next state of the algorithm would be *S11* and therefore the calculation would end. If the defect was larger than the desired value, the next state would be *S6* and the iteratioun would complete normally and loop from the state *S10* to *S1*.

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

State	WFL Code	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	CS
S0	R1 ← x1; R2 ← x2	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	36'3C000000
S1	R3 ← R1 × R2 (1) R4 ← R2 × 2 (2)	0	0	1	1	0	0	0	0	0	1	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	36'3A200000
S2	R3 ← R3 + R4 (1) R4 ← R1 - R4 (2) R5 ← R3 × 3 (3)	0	0	1	1	1	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	0	1	0	36'3A240000
S3	R3 ← R3 - R2 (1) R6 ← R6 - 2 (2) R5 ← R3 × (23) (3) R7 ← F200	0	0	1	1	0	0	1	1	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	0	1	0	36'3A310000
S4	R3 ← R3 - 1 (1) R4 ← R7 - R4 (2) R8 ← R4 + 1 (3) R6 ← F200	0	0	1	1	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	1	0	1	0	0	0	0	36'3A512000
S5	R6 ← R6 - R5 (1) R9 ← 1/R4 (1)	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	36'3A210000
S6	R3 load from the value when data is available R6 ← R9 × (2) (1) R7 ← R9 × (1) (2) R5 ← R5 × R6 (3)	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	36'31000000
S7	R6 ← R3 × R6 (1) R4 ← R4 × R6 (2) R5 ← R3 × R6 (3) R7 ← R3 × R6 (4)	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	36'31000000
S8	R3 ← R3 + R4 (1) R5 ← R5 + R7 (2) R2 ← R2 + R5 (2)	0	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	36'3C000000
S9	R3 ← R3 + R5 (1) R2 ← R2 + R5 (2)	0	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	36'3C000000
S10	R3 ← R3 + R5 (1) R2 ← R2 + R5 (2)	0	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	36'3C000000
S11		0	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	36'3C000000

5.3 Simulation results

The test bench for simulation was made using Cocotb [1] with the Verilator [2] as a simulator. The result 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 for this design is currently 20 MHz.

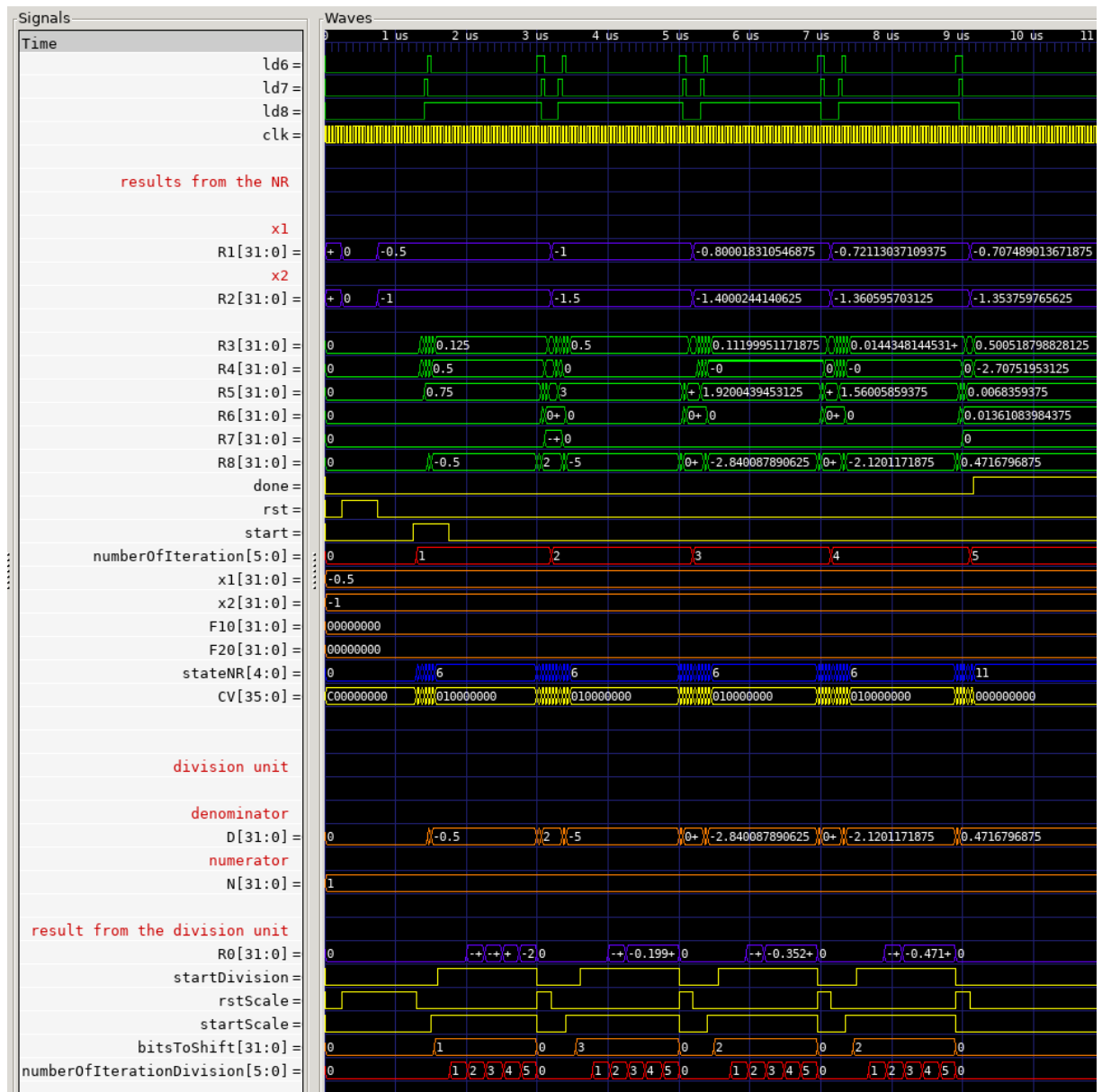


Figure 5 - 4 The whole Verilog simulation of a simple Newton-Raphson (NR) algorithm. The result is may be seen in registers R1 and R2 after the fifth iteration of the algorithm.

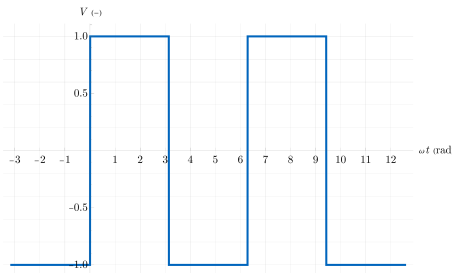
6 Selective Harmonic Elimination

6.1 Theory

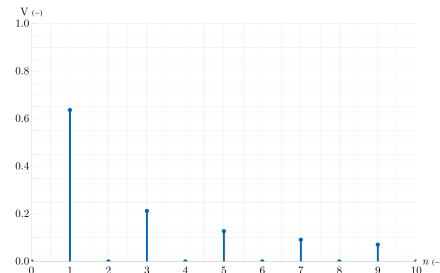
The original theory for Selective Harmonic Elimination was developed in [8, 9] and later adopted by many researchers and adopted for multiple voltage inverter topologies. Nowadays the strategy is mainly used in traction applications after control and modulation strategies for start up state end and the reference voltage for the drive is high enough so the six step output voltage is utilized. However general six step output signal yields high order harmonics. When the motor is powered by these high order voltage harmonics, the current with high order harmonics (excluding triplen harmonics, because the symmetric 3 phase motor is considered) is then observed. This current harmonics cause undesirable current ripple, torque ripple and losses [10] which decrease the efficiency of the drive.

To control the output voltage and reduce unwanted harmonics the Selective Harmonic Elimination (SHE) technique may be employed. The elimination is based on generating the output voltage by switching the the components at certain angles, thus generating waveform with number of pulses to which corresponds the number of eliminated harmonics. The calculation is based on the calculation of fourier coefficients. The principle has been modified for different types of converters, such as multilevel, H-bridge converters or generic Voltage Source Inverters (VSI). In this paper, the regular two level VSI is considered.

The considered inverter voltage waveform is depicted in Figure 6 - 1a. The harmonic analysis of the generic waveform is depicted in Figure 6 - 1b. Note that in a 3 phase symmetrical system the triplen harmonics would be eliminated as well.



(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

As previously mentioned, the method is based on a Fourier coefficient analysis. When the odd quarter-wave symmetry of the waveform is assumed, 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.

$$a_n = 0, \quad (6 - 1)$$

$$b_n = \frac{1}{T} \int_0^T x(n\omega t) \sin(\omega t) d\omega t, \quad (6 - 2)$$

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

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

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

The function $x(\omega t)$ describes the output voltage pulse value normalized to a DC link voltage. The Equation 6 - 2 may then be rewritten using the substitution of ωt by the angles α which also describe the output waveform dependent on radians and that the function $x(\alpha)$ yields 1 when the output voltage pulse is positive and -1 when negative. The rewritten equation 6 - 2 when assuming quarter-wave symmetry

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

Where M is number of pulses in half periode of the output signal. When assuming that the integral is calculated for states when the $x(\alpha_k)$ is 1 or -1 , the function may be replaced by a constant, thus the integral calculation is quite simple.

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

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

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

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

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

Whereas 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 to be solved to eliminate one harmonics is as follows.

$$\begin{aligned} V_1 &= b_1 = \frac{4}{\pi} [\cos(\alpha_1) - \cos(\alpha_2)], \\ V_5 &= b_5 = \frac{4}{5\pi} [\cos(5\alpha_1) - \cos(5\alpha_2)]. \end{aligned} \quad (6 - 8)$$

The $V_1 = b_1$, $V_5 = b_5$ are the amplitudes of 1st, respectively 5th harmonics. Where for the elimination of the 5th harmonics must be true that $b_5 = 0$. So the set of equations 6 - 8 may be simplified as set of Equations 6 - 9.

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

The solution of the Equations 6 - 9 is not trivial as they are nonlinear. There may be various methods how to solve the problem, such as Genetic Algorithms [11, 12, 13] or algebraic methods [14, 15]. One of the well known used algebraic methods is Newton-Raphson (NR) algorithm [16]. On this paper, the solution is obtained solely by using NR algorithm. The problem of this method is that it is required to set the initial conditions wellm otherwise the solution may not be found. On the other hand, the Genetic Algorithms need to set the initial values as well, but often random numbers from a predefined intervals are used.

For real time systems, the approach of SHE 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 FPGA may be more often utilized to calculate the solution. The caclulation may be highly paralelized and optimized to obtain the solution in near real time. In following sections the prototype implementation in Python and final implementaion in Verilog is presented.

6.2 Simplification for Verilog and High level implementation

When implementing the solution in computational software, such as, Wolfram Mathematica, the optimization of the algorithm is very often not needed. However, when implementing the algorithm to a FPGA the higher level constructs are not easily available, so the simplification of the algorithm must be done. For clarity and prototyping purposes, the Python implementation optimization level is lower, than for the Verilog. In this section, the simplified algorithm of a NR aglorithm is presented.

The equation for eliminating the 5th harmonics may be written as

$$\begin{aligned} F_1^i &= \cos(\alpha_1) - \cos(\alpha_2), \\ F_2^i &= \cos(5\alpha_1) - \cos(5\alpha_2), \\ \text{where } F_1^0 &= m \frac{\pi}{4}, F_2^0 = 0. \end{aligned} \quad (6 - 10)$$

Thus the Jakobian matrix is

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

Where i is the index of the iteration of the algorithm. Next the inverted Jakobian matrix is needed for further calculations.

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

From the inverted Jakobian matrix it can be seen, that it can be easily calculated by division of components by the determinant, which can be expressed as

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

Next, the defect ΔF^i may be calculated

$$\begin{aligned} \Delta F_1^i &= F_1^0 - F_1^i, \\ \Delta F_2^i &= F_2^0 - F_2^i. \end{aligned} \quad (6 - 14)$$

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

$$\Delta\alpha^i = \mathbf{J}^{-1,i} \Delta\mathbf{F}^i, \quad (6 - 15)$$

thus rewritten in components notation suitable for the Verilog implementation

$$\begin{aligned} \Delta\alpha_1^i &= \mathbf{J}_{00}^{-1,i} \Delta F_1^i + \mathbf{J}_{01}^{-1,i} \Delta F_2^i, \\ \Delta\alpha_2^i &= \mathbf{J}_{10}^{-1,i} \Delta F_1^i + \mathbf{J}_{11}^{-1,i} \Delta F_2^i. \end{aligned} \quad (6 - 16)$$

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

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

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

It is important to mention, that for the NR algorithm to work correctly, the suitable initial values F_1^0 and F_2^0 must be well chosen before the algorithm starts.

When eliminating the 5th harmonic in a settings, where $m = 1$, the initial values of $F_2^0 = 0.08726$ rad and $F_2^0 = 1.3439$ rad yield suitable results.

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

6.3 High level implementation

The algorithm was for rapid prototyping purposes implemented using Python. The script incorporates changing the modulation index at the start of the python simulation, thus enables generating values which then may 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.

```

1 import math
2 import argparse # for parsing command line arguments
3
4 # colorama for colors, easier than init class, maybe later
5 # source: https://github.com/tartley/colorama
6 from colorama import init as colorama_init
7 from colorama import Fore
8 from colorama import Style
9
10 colorama_init(autoreset=True) # autoreset color on new line
11
12 # class with additional styles
13 class style:
14     BOLD = '\033[1m'
15     UNDERLINE = '\033[4m'
16     END = '\033[0m'
```

```

17
18 argParser = argparse.ArgumentParser() # new object
19 argParser.add_argument("-mi", "--modulationIndex", help="set the modulation
    index 0-1") # adding argument
20 args = argParser.parse_args() # parsing args
21 modulationIndex = args.modulationIndex
22
23 # Set the desired modulation index
24 if not modulationIndex:
25     print()
26     print(style.BOLD+Fore.RED + "You did not specify the modulation index
    with mi command, specify it now:\n" + style.END)
27     modulationIndex = input()
28
29 print("You have specified the modulation index: " + modulationIndex + ".\n"
    )
30
31 modulationIndex = float(modulationIndex)
32 totalNumberOfIterations = 10
33 f10 = modulationIndex * 0.7853981 # modulationIndex * pi/4
34 f20 = 0
35 x10 = 0.0872664 # 5 degree
36 x20 = 1.3439035 # 77 degree
37
38 x1 = x10
39 x2 = x20
40
41 # main NR-LOOP
42 for numberOfIteration in range(totalNumberOfIterations):
43     prepDeltaF1 = math.cos(x1) - math.cos(x2)
44     deltaF1 = f10 - prepDeltaF1
45
46     prepDeltaF2 = math.cos(5*x1) - math.cos(5*x2)
47     deltaF2 = f20 - prepDeltaF2
48
49     prepJ11 = math.sin(x1)
50     prepJ01 = math.sin(x2)
51     prepJ10 = 5 * math.sin(5*x1)
52     prepJ00 = 5 * math.sin(5*x2)
53
54
55     prepDet1 = prepJ10 * prepJ01
56     prepDet2 = 5 * prepJ11 * math.sin(5*x2)
57
58     prepDet = prepDet1 - prepDet2
59
60     divDet = 1 / prepDet
61

```

```

62     jInv00 = divDet * prepJ00
63     jInv01 = divDet * - prepJ01
64     jInv10 = divDet * prepJ10
65     jInv11 = divDet * - prepJ11
66
67
68     deltaX1 = (jInv00 * deltaF1) + (jInv01 * deltaF2)
69     deltaX2 = (jInv10 * deltaF1) + (jInv11 * deltaF2)
70
71     x1 = x1 + deltaX1
72     x2 = x2 + deltaX2
73
74     print(Fore.CYAN + "numberOfIteration: " + str(numberOfIteration) +
75           style.END)
76
77 # End of the main NR-LOOP
78
79 print(Fore.GREEN + "x1: " + str(x1) + style.END)
80 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 calculation algorithm for SHE, mathematically expressed in the section *Simplification for Verilog and High level implementation*.

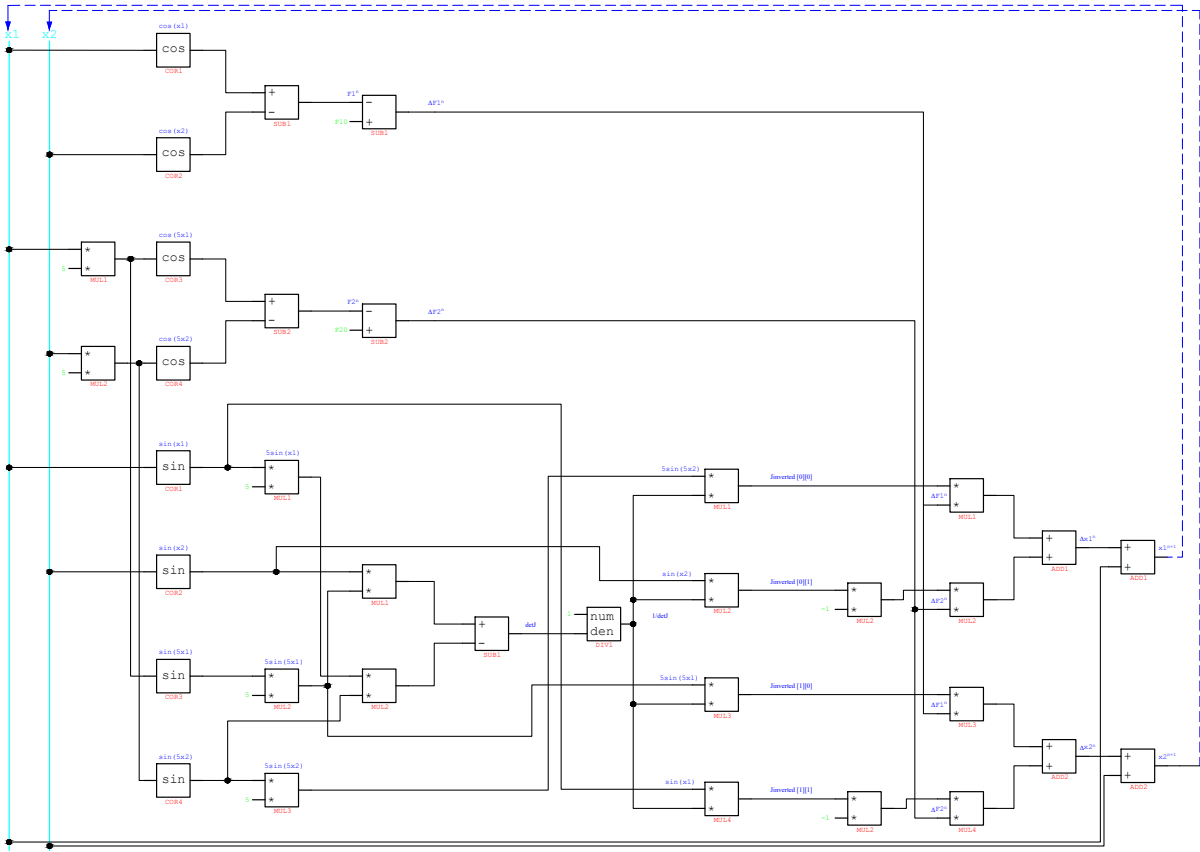


Figure 6 - 2 Block Diagram of the Selective Harmonic Elimination (SHE) using Newton-Raphson algorithm.

6.4.2 Top module design

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

As for every design presented, the units utilize the $Q32.15$ fixed point format for it's computational units and registers, the exception being multiplier computational units, which by the principle of multiplication use format $Q64.30$ for the results. When the multiplication results are passed to registers, the values are rounded back to globally used format.

The design is depicted on Figure 6 - 3.

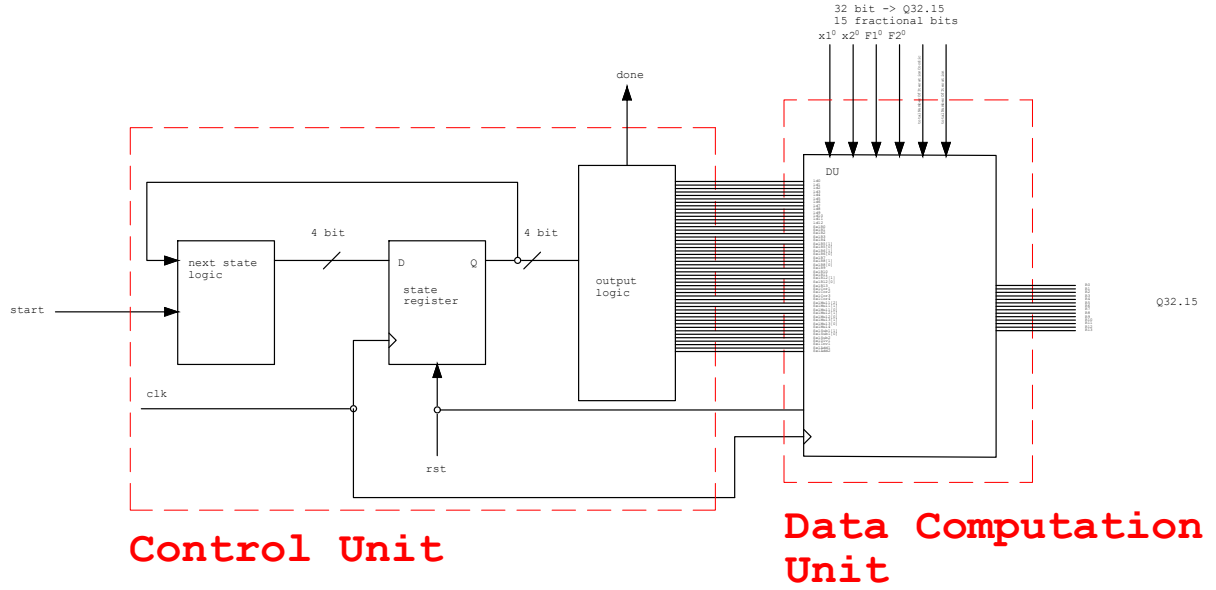


Figure 6 - 3 Top module design for the Selective Harmonic Elimination unit (SHE).

6.4.3 Allocation and Timing

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

The Verilog implementation consists of totally 13 states $S0-S12$. Through states $S1-S11$ the NR algorithm iterates to calculate the ending results. The state $S0$ is a starting state after resetting the unit and state $S12$ is ending state, which is reached after the successfull 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 sobmodule algorithms are not concern of this part and are implicitly accepted as a part of the SHE module algorithm.

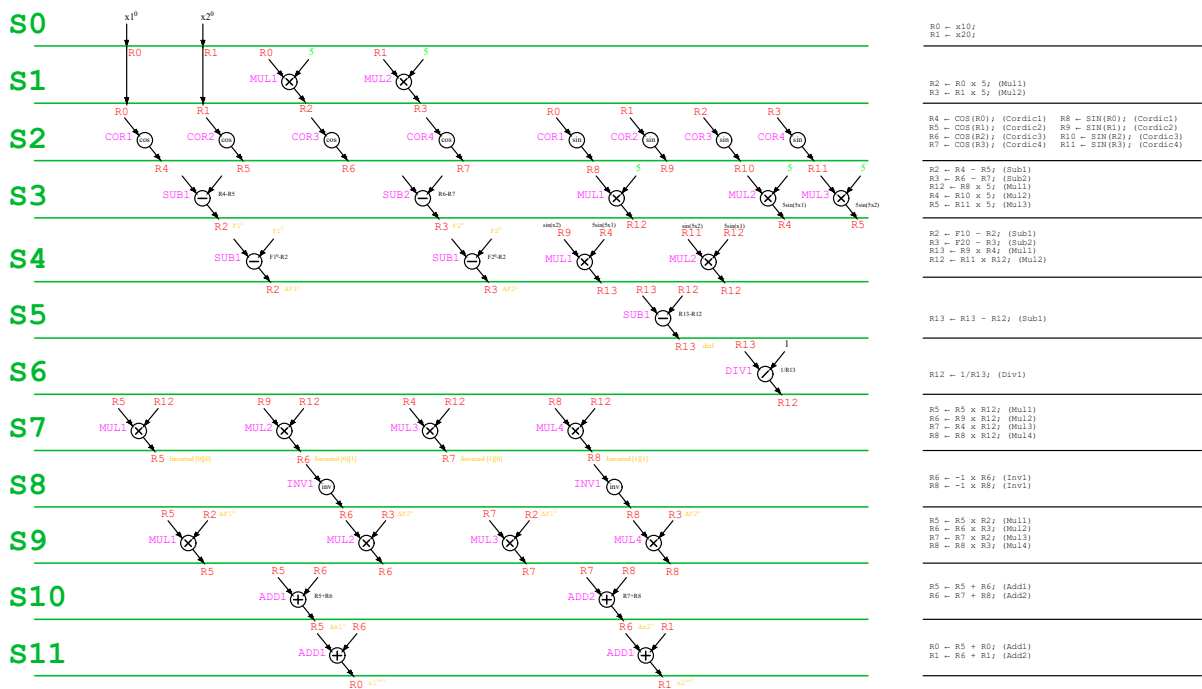


Figure 6 - 4 Allocation and Timing diagram for the Data Path Unit part of Selective Harmonic Elimination (SHE) module.

6.4.4 Data Path Unit

As can be observed from the Figure 6 - 5 the Data Path unit for solving the transcendental 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 inverter unit, which is implemented directly in the registers logic.

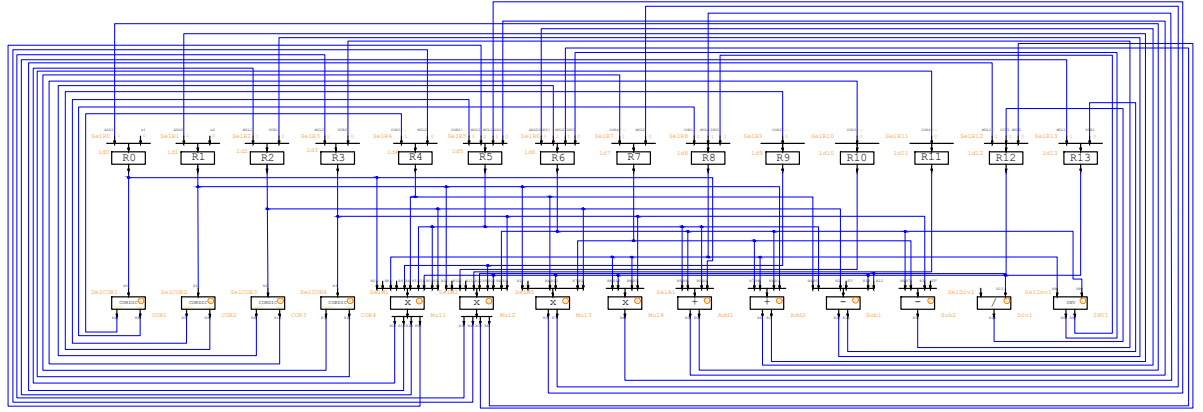


Figure 6 - 5 Register transfer level (RTL) scheme of the Selective Harmonic Elimination Data Path Unit.

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 heightened.

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

Inst	Inst Code	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275	276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300	301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325	326	327	328	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350	351	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375	376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392	393	394	395	396	397	398	399	400	401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425	426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450	451	452	453	454	455	456	457	458	459	460	461	462	463	464	465	466	467	468	469	470	471	472	473	474	475	476	477	478	479	480	481	482	483	484	485	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500	501	502	503	504	505	506	507	508	509	510	511	512	513	514	515	516	517	518	519	520	521	522	523	524	525	526	527	528	529	530	531	532	533	534	535	536	537	538	539	540	541	542	543	544	545	546	547	548	549	550	551	552	553	554	555	556	557	558	559	560	561	562	563	564	565	566	567	568	569	570	571	572	573	574	575	576	577	578	579	580	581	582	583	584	585	586	587	588	589	590	591	592	593	594	595	596	597	598	599	600	601	602	603	604	605	606	607	608	609	610	611	612	613	614	615	616	617	618	619	620	621	622	623	624	625	626	627	628	629	630	631	632	633	634	635	636	637	638	639	640	641	642	643	644	645	646	647	648	649	650	651	652	653	654	655	656	657	658	659	660	661	662	663	664	665	666	667	668	669	670	671	672	673	674	675	676	677	678	679	680	681	682	683	684	685	686	687	688	689	690	691	692	693	694	695	696	697	698	699	700	701	702	703	704	705	706	707	708	709	710	711	712	713	714	715	716	717	718	719	720	721	722	723	724	725	726	727	728	729	730	731	732	733	734	735	736	737	738	739	740	741	742	743	744	745	746	747	748	749	750	751	752	753	754	755	756	757	758	759	760	761	762	763	764	765	766	767	768	769	770	771	772	773	774	775	776	777	778	779	780	781	782	783	784	785	786	787	788	789	790	791	792	793	794	795	796	797	798	799	800	801	802	803	804	805	806	807	808	809	810	811	812	813	814	815	816	817	818	819	820	821	822	823	824	825	826	827	828	829	830	831	832	833	834	835	836	837	838	839	840	841	842	843	844	845	846	847	848	849	850	851	852	853	854	855	856	857	858	859	860	861	862	863	864	865	866	867	868	869	870	871	872	873	874	875	876	877	878	879	880	881	882	883	884	885	886	887	888	889	890	891	892	893	894	895	896	897	898	899	900	901	902	903	904	905	906	907	908	909	910	911	912	913	914	915	916	917	918	919	920	921	922	923	924	925	926	927	928	929	930	931	932	933	934	935	936	937	938	939	940	941	942	943	944	945	946	947	948	949	950	951	952	953	954	955	956	957	958	959	960	961	962	963	964	965	966	967	968	969	970	971	972	973	974	975	976	977	978	979	980	981	982	983	984	985	986	987	988	989	990	991	992	993	994	995	996	997	998	999	1000	1001	1002	1003	1004	1005	1006	1007	1008	1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023	1024	1025	1026	1027	1028	1029	1030	1031	1032	1033	1034	1035	1036	1037	1038	1039	1040	1041	1042	1043	1044	1045	1046	1047	1048	1049	1050	1051	1052	1053	1054	1055	1056	1057	1058	1059	1060	1061	1062	1063	1064	1065	1066	1067	1068	1069	1070	1071	1072	1073	1074	1075	1076	1077	1078	1079	1080	1081	1082	1083	1084	1085	1086	1087	1088	1089	1090	1091	1092	1093	1094	1095	1096	1097	1098	1099	1100	1101	1102	1103	1104	1105	1106	1107	1108	1109	1110	1111	1112	1113	1114	1115	1116	1117	1118	1119	1120	1121	1122	1123	1124	1125	1126	1127	1128	1129	1130	1131	1132	1133	1134	1135	1136	1137	1138	1139	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149	1150	1151	1152	1153	1154	1155	1156	1157	1158	1159	1160	1161	1162	1163	1164	1165	1166	1167	1168	1169	1170	1171	1172	1173	1174	1175	1176	1177	1178	1179	1180	1181	1182	1183	1184	1185	1186	1187	1188	1189	1190	1191	1192	1193	1194	1195	1196	1197	1198	1199	1200	1201	1202	1203	1204	1205	1206	1207	1208	1209	1210	1211	1212	1213	1214	1215	1216	1217	1218	1219	1220	1221	1222	1223	1224	1225	1226	1227	1228	1229	1230	1231	1232	1233	1234	1235	1236	1237	1238	1239	1240	1241	1242	1243	1244	1245	1246	1247	1248	1249	1250	1251	1252	1253	1254	1255	1256	1257	1258	1259	1260	1261	1262	1263	1264	1265	1266	1267	1268	1269	1270	1271	1272	1273	1274	1275	1276	1277	1278	1279	1280	1281	1282	1283	1284	1285	1286	1287	1288	1289	1290	1291	1292	1293	1294	1295	1296	1297	1298	1299	1300	1301	1302	1303	1304	1305	1306	1307	1308	1309	1310	1311	1312	1313	1314	1315	1316	1317	1318	1319	1320	1321	1322	1323	1324	1325	1326	1327	1328	1329	1330	1331	1332	1333	1334	1335	1336	1337	1338	1339	1340	1341	1342	1343	1344	1345	1346	1347	1348	1349	1350	1351	1352	1353	1354	1355	1356	1357	1358	1359	1360	1361	1362	1363	1364	1365	1366	1367	1368	1369	1370	1371	1372	1373	1374	1375	1376	1377	1378	1379	1380	1381	1382	1383	1384	1385	1386	1387	1388	1389	1390	1391	1392	1393	1394	1395	1396	1397	1398	1399	1400	1401	1402	1403	1404	1405	1406	1407	1408	1409	1410	1411	1412	1413	1414	1415	1416	1417	1418	1419	1420	1421	1422	1423	1424	1425	1426	1427	1428	1429	1430	1431	1432	1433	1434	1435	1436	1437	1438	1439	1440	1441	1442	1443	1444	1445	1446	1447	1448	1449	1450	1451	1452	1453	1454	1455	1456	1457	1458	1459	1460	1461	1462	1463	1464	1465	1466	1467	1468	1469	1470	1471	1472	1473	1474	1475	1476	1477	1478	1479	1480	1481	1482	1483	1484	1485	1486
------	-----------	---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------

Conclusion

This paper presents FPGA module suitable for solving the SHE algorithm in near real-time. The module consists of two other submodules which are also presented in this paper. These submodules are units for calculating the division of two arbitrary values and a CORDIC unit, suitable for calculating *sinus* and *cosinus* functions.

The goal of this paper was to make investigate and design speed optimized modules, which would be capable of near real-time calculations. Outcomes of this paper may be in the future used as a starting point for other researches of designing the modules for controlling the electric drives or for 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-Levelled 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
VSI	Voltage Source Converter