



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	12
4.1	Theory	12
4.1.1	Example of calculation	14
4.2	Python Implementation	14
4.3	IP Block Design	17
4.3.1	Top module design	17
4.3.2	Allocation and Timing	18
4.3.3	Data Path Module	19
4.3.4	Control Unit	22
4.4	Simulation results	22
5	Simple set of nonlinear equations solved by a Newton-Raphson algorithm using custom circuit implementation	26
5.1	Theory	26
5.2	IP Block Design	27
5.2.1	Top module design	27
5.2.2	Allocation and Timing	28
5.2.3	Data Path Unit	29
5.2.4	Control Unit	31
5.3	Simulation results	31
6	Selective Harmonic Elimination	33
6.1	Theory	33
6.2	IP Block Design	33
6.2.1	Top module design	33
6.2.2	Allocation and Timing	33
6.2.3	Data Path Unit	33
6.2.4	Control Unit	33

Conclusion	34
References	35
Appendix A List of Symbols and Abbreviations	36
A.1 List of abbreviations.....	36
A.2 List of symbols	36

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).	9
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).	10
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).	11
4 - 1	Top module design for the CORDIC module block design.	18
4 - 2	Allocation and timing diagram for the Data Path Unit part of the CORDIC IP.	19
4 - 3	Register transfer level RTL scheme of the CORDIC IP Data Path Unit IP.	20
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.	23
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.	24
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.	24
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.	25
5 - 1	Top module design for the simple Newton-Raphson (NR) calculation unit module block design.	28
5 - 2	Allocation and timing diagram for the Data Path Unit part of the simple (NR) module. ...	29
5 - 3	Register Transfer Level (RTL) scheme of the Data Path Unit part of the simple Newton-Raphson (NR) calculation IP.	30
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.	32

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. ...	22
5 - 1	Control signal encoding table for instructions to be processed by the simple Newton-Raphson (NR) algorithm solve Module.	31
6 - 1	Control signal encoding table for instructions to be processed by the simple Newton-Raphson (NR) algorithm solve Module.	33

1 Introduction

This is the introduction.

2 Notes on all of the circuit designs in Verilog

All of the designs 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, as it entails 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 [**amd-xilinx-vivado-divider-ip-block**] or feature low performance [3].

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. [3]

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 unknown variables. When the initial variables 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. [3] The formula for calculating the initial value eq. 3 - 1 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 and D is the denominator value for calculating the expression N/D .

Because the fixed point number format $Q32.15$ is used, the fractional numbers in equation 3 - 1 are rounded to 2.8229 (32'sb00000000000000010_110100101011000 in binary) and 1.8819 (32'sb00000000000000001_111000011100101 in binary) respectively.

After the initial value x_0 is calculated, the NR algorithm is performed. The idea for 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. For the NR algorithm the function with root is $1/D$ is essential. There may be many functions, which root is the searched value $1/D$ but the most trivial is eq. 3 - 2.

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

For the derivative at the point of x_i then applies eq. 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 fig. 3 - 1. Thanks to this wrapper it is possible to test the created modules with Verilog Testbench, Verilator [2] or Cocotb [1].

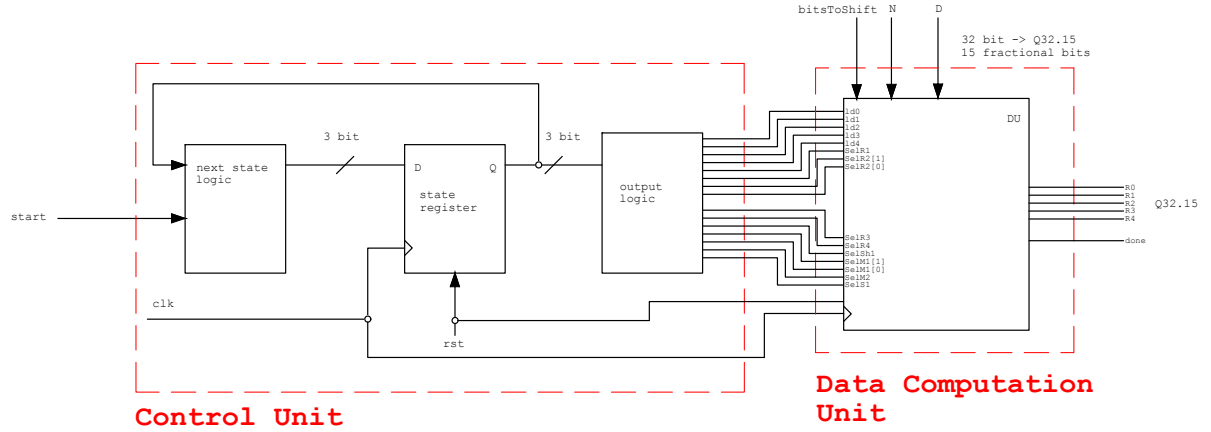


Figure 3 - 1 Top module design for the division unit module block design.

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 consists of nine steps. The first four steps are used for calculating the initial value of x_0 as described in the equation 3 - 1. The steps $S4$ to $S8$ are for calculating the next search value of x_{i+1} , the root of the equation 3 - 2 so 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 met, such as reaching a specified number of iterations.

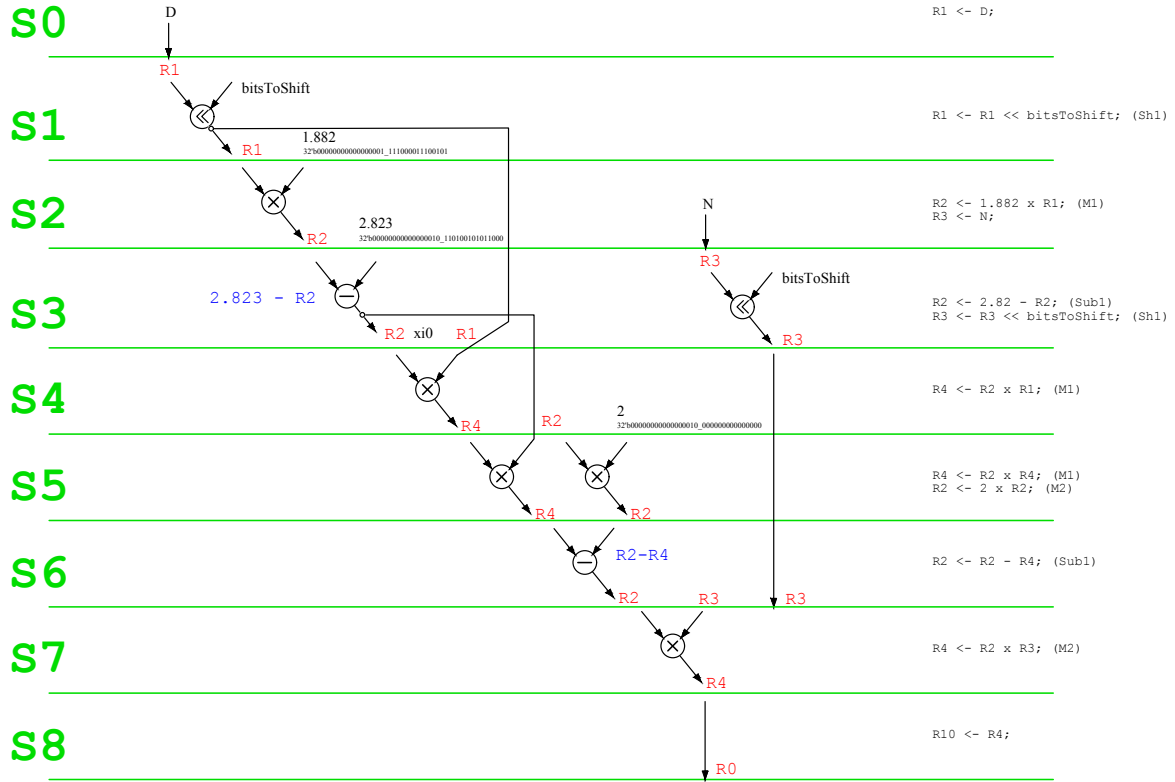


Figure 3 - 2 Allocation and timing diagram for the Data Path Unit part of the division module.

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 with the control signal labeled as CS . The encoding table with the labels which corresponds 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 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.

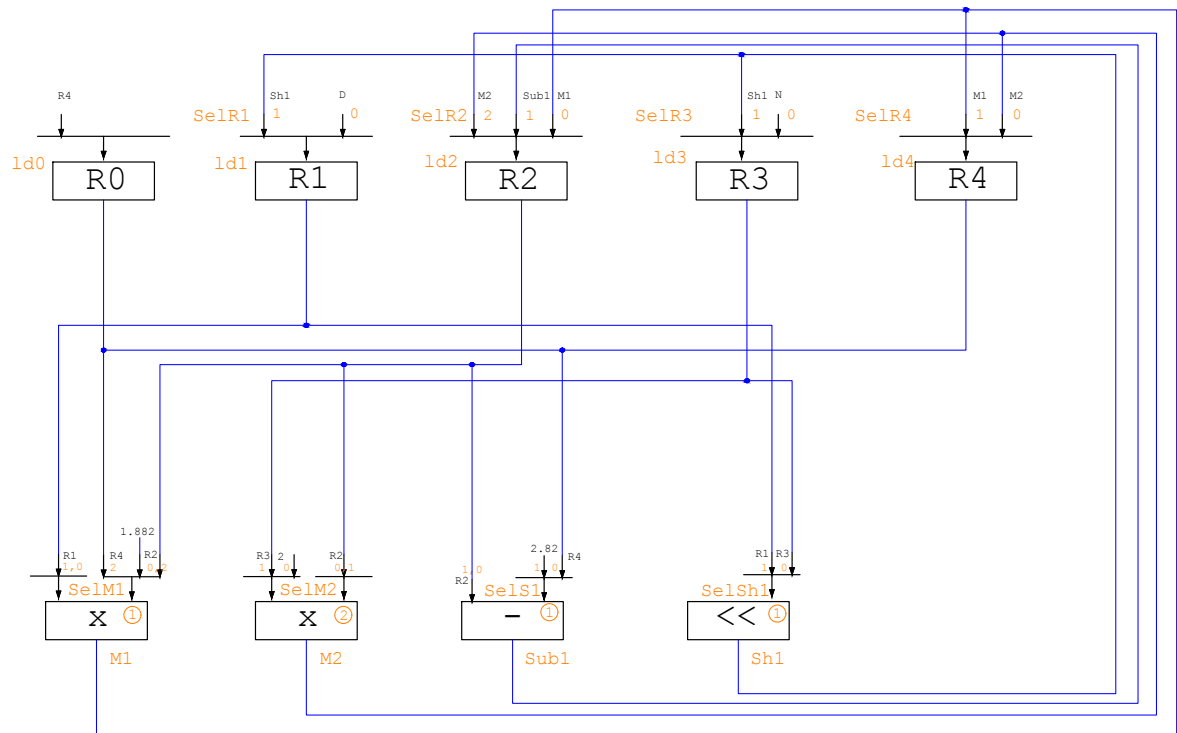


Figure 3 - 3 Register Transfer Level (RTL) scheme of the Data Path Unit part of the division module.

3.2.4 Control Unit

The signals from Control Unit to Data Path Module are encoded in the CS signal. The CS signal with the corresponding instructions for the steps S_0 – S_8 of the FSM is presented in the table 3 - 1. For cleaner code, the signal is passed to the Control Unit in the hexadecimal format.

The number of the iteration is also set in the Control Unit. The value is used in this module to determine the stop condition of the calculation.

As stated in the *Allocation and Timing* section, after the step S_8 , the FSM restarts at the state S_4 with new x_i values to be used in the current iteration. This jump is not depicted in the table 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 its 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 else
```

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.

As for the simulation output it can be stated, that the module works correctly for positive and negative numbers of fixed point format *Q32.15*.

The algorithm used in this module is able to calculate the proper result in much less clock cycles than the full division algorithm used in the division module in the package [3].

Thus the presented module may be used as a submodule in more complex modules.

VCD simulation output waveforms are depicted on the following Figures. The simulations were conducted for arbitrary selected N and D . The clock frequency was set 250 MHz. Pseudocode Verilog snippet for the test bench is present 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'b00000000100110000_0000100000000000; D=32'
   b11111111111111111111_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.

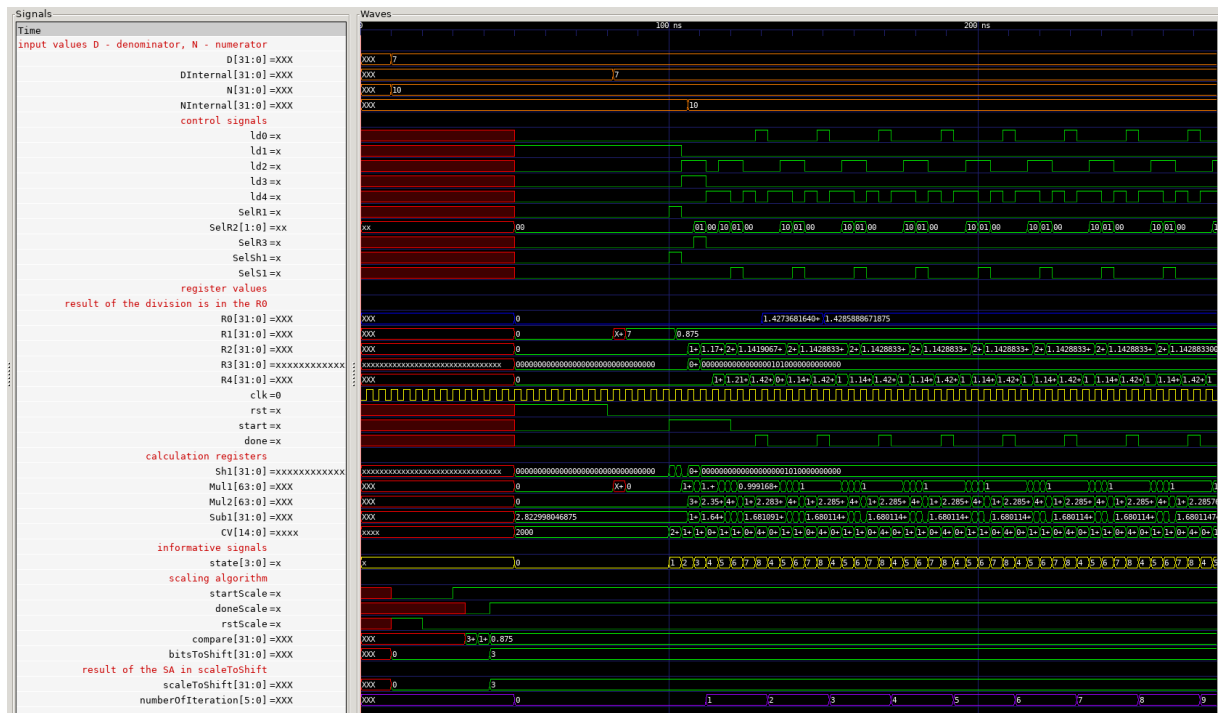


Figure 3 - 4 Selected signals of simulation of division $N/D = 10 / 7$. The correct result in R0 is obtained after two iterations (reg numberOfIterations).

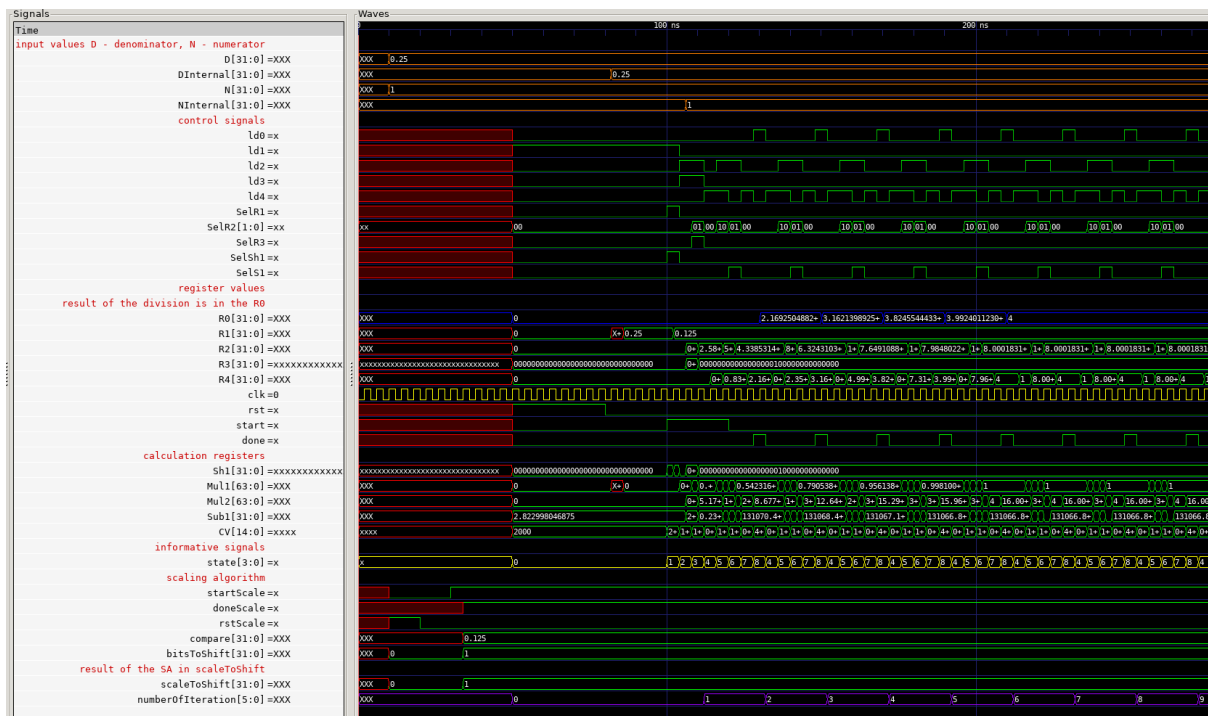


Figure 3 - 5 Selected signals of simulation of division $N/D = 1 / 0.25$. The correct result in R0 is obtained after five iterations (reg numberOfIterations).

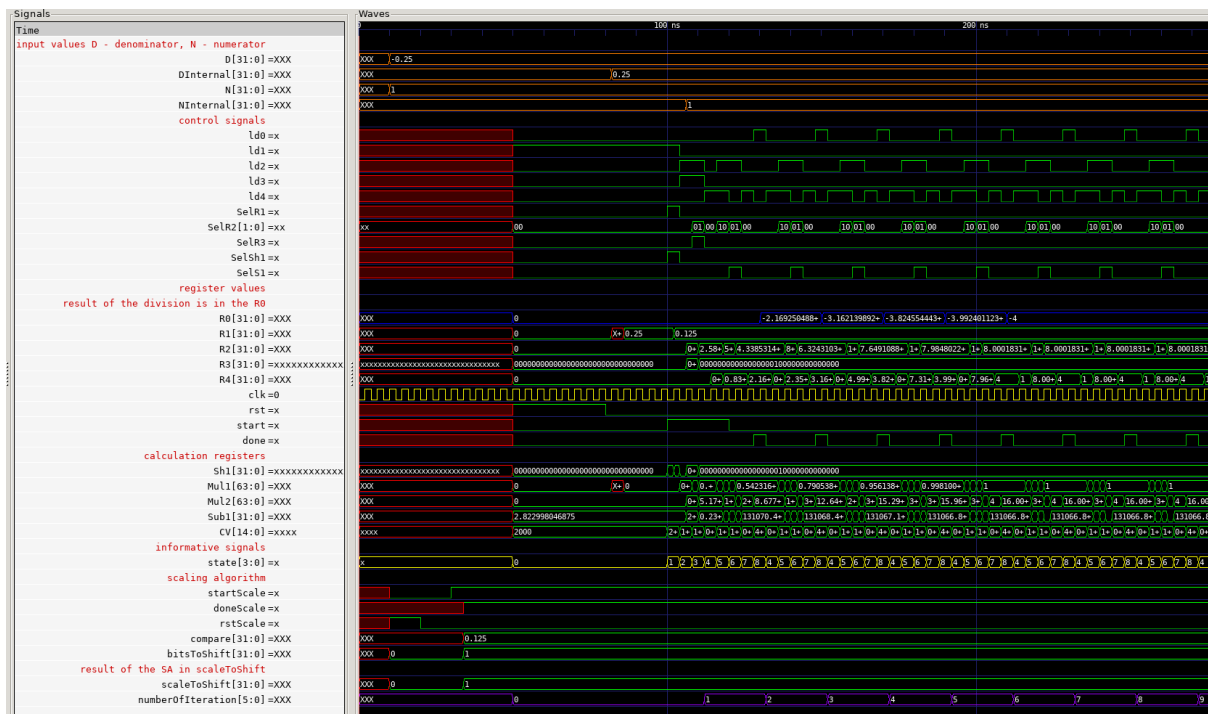


Figure 3 - 6 Selected signals of simulation of division $N/D = 1 / (-0.25)$. The correct result in R0 is obtained after five iterations (reg numberOfIterations).

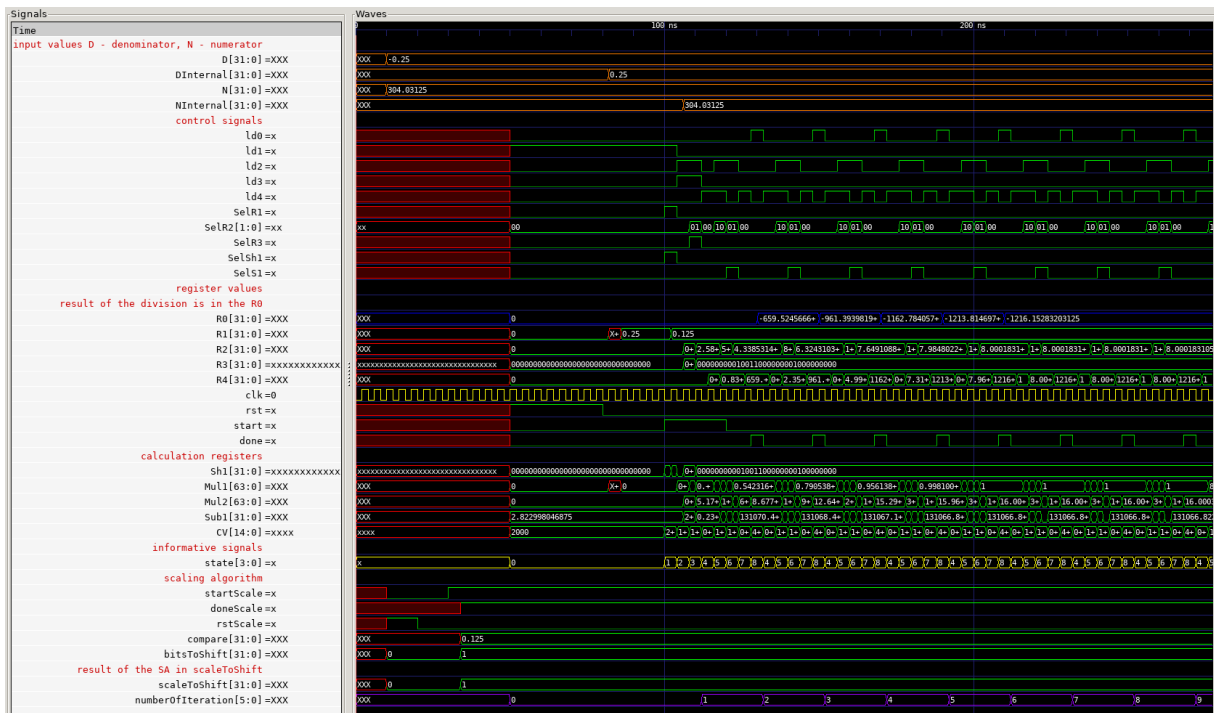


Figure 3 - 7 Selected signals of simulation of division $N/D = 304.03215 / (-0.25)$. The correct result in R0 is obtained after five iterations (reg numberOfIterations).

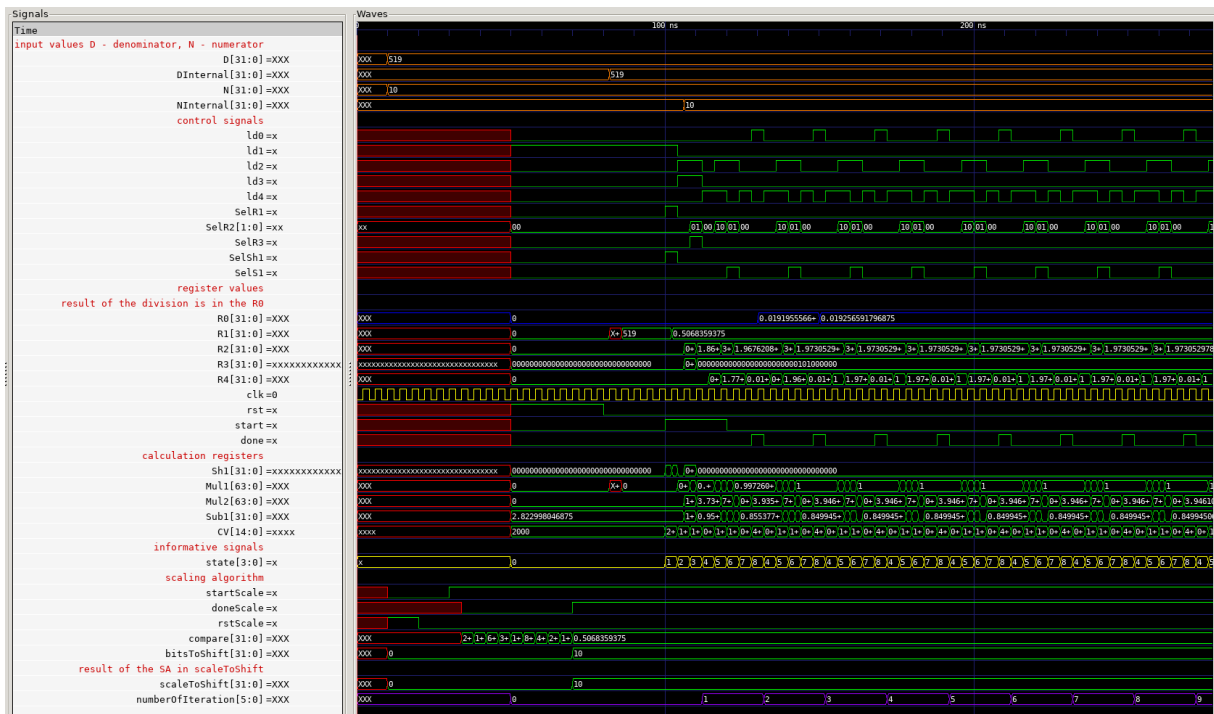


Figure 3 - 8 Selected signals of simulation of division $N/D = 10 / (519)$. The correct result in R0 is obtained after two iterations (reg numberOfIterations).

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. [4] 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 [5]. This algorithm computes a coordinate conversion between rectangular (x, y) and polar (R, θ) coordinates. The algorithm was then generalized by Walther in [6] 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 destructurized in the base angles, for which the equation 4 - 3 is true. In this example the is destructurized 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 tradeof 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 tanges 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.

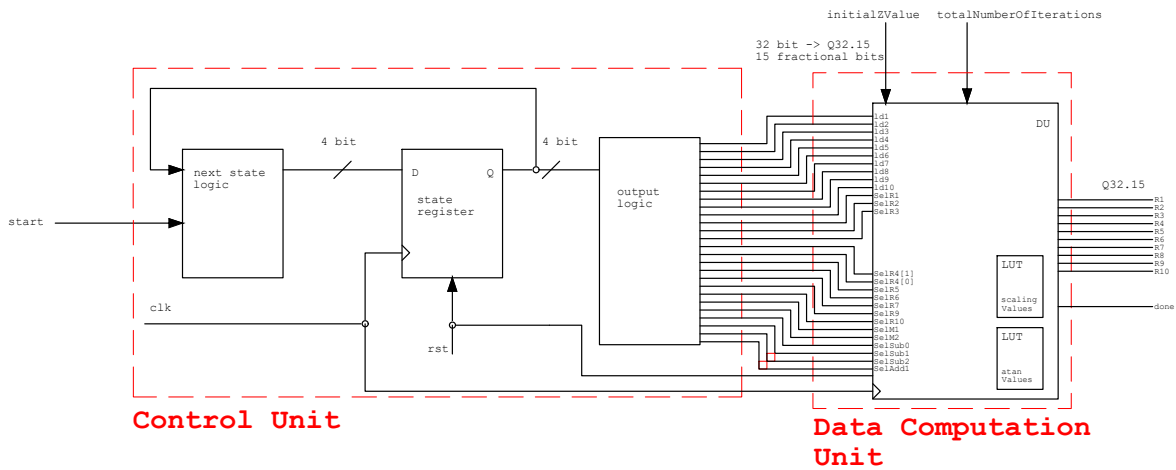


Figure 4 - 1 Top module design for the CORDIC module block design.

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.

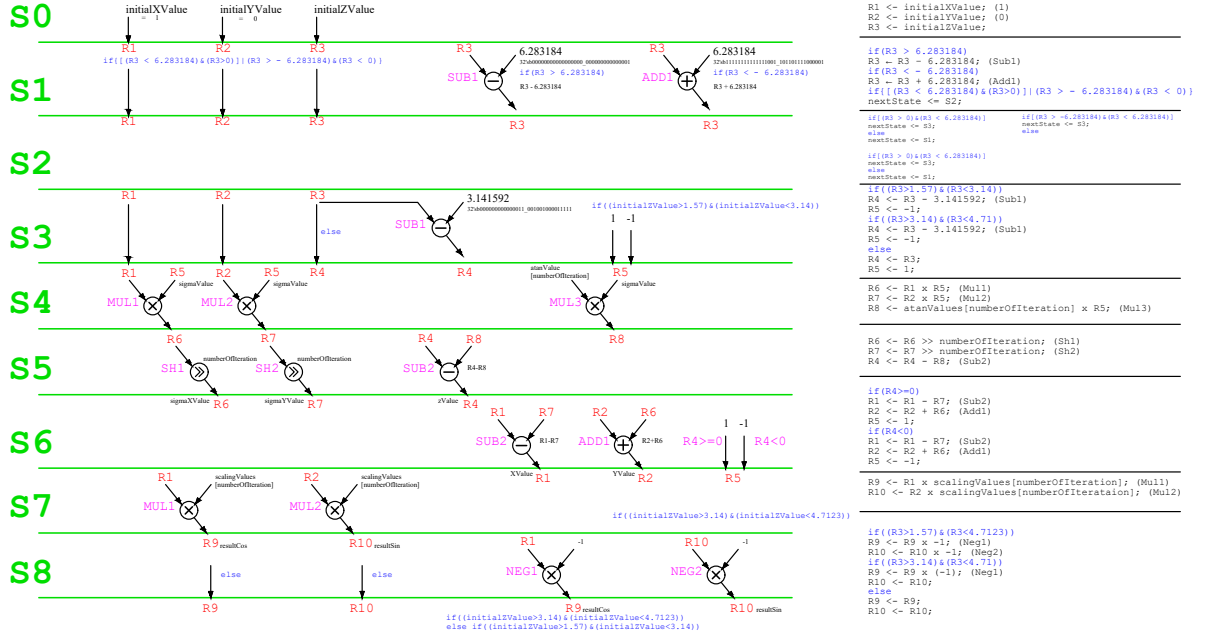


Figure 4 - 2 Allocation and timing diagram for the Data Path Unit part of the CORDIC IP.

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 **SelR_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.

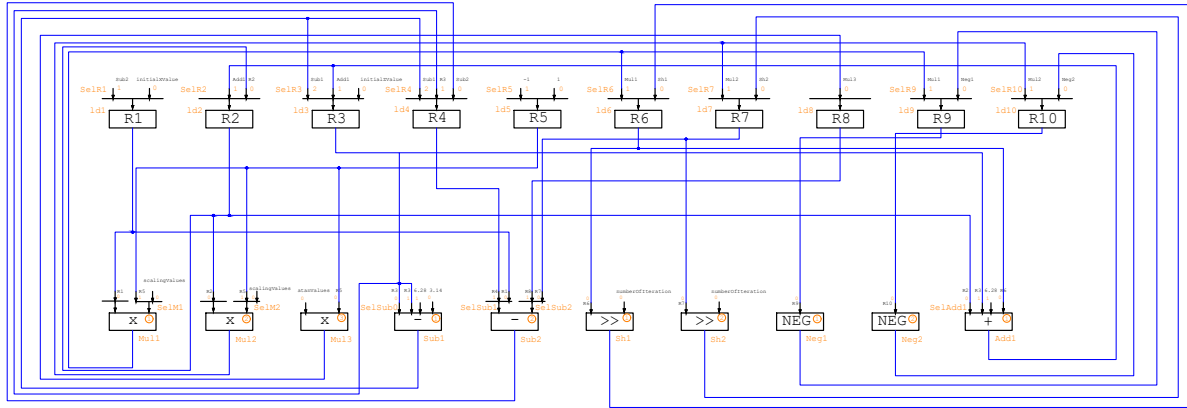


Figure 4 - 3 Register transfer level RTL scheme of the CORDIC IP Data Path Unit IP.

```

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_000000011111111; //
    0.007812341060101111
18     4'b1000: returnValue = 32'sb000000000000000000_000000011111111; //
    0.007812341060101111
19     4'b1001: returnValue = 32'sb000000000000000000_000000001111111; //
    0.0019531225164788188
20     4'b1010: returnValue = 32'sb000000000000000000_000000000111111; //
    0.0009765621895593195
21     4'b1011: returnValue = 32'sb000000000000000000_000000000011111; //
    0.0004882812111948983
22     default: returnValue = 32'sb000000000000000000_000000000000000; // 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_000000000000000; //
            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: returnValue <= 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 alogrithm 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	
	R0 ← totalNumberOfIterations; R1 ← initialXValue; R2 ← initialYValue; R3 ← initialZValue; if(R3 >= 283184) R3 ← R3 - 6283184; (Sub1)	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	27'b700000	
S1	if(R3 <= 6283184) R3 ← R3 + 6283184; (Add1) if((R3 >= 283184 & (R3 <= 0)) (R3 <= -6283184 & (R3 < 0))) → nextState <= S2; else → nextState <= S1;	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	27'b100000
S2	if(R3 > 0 & (R3 <= 283184)) → nextState <= S3; CS = 0; else → nextState <= S1; if(R3 > 0 & (R3 <= 283184)) → nextState <= S3; CS = 0; else → nextState <= S1; if(R3 <= 283184 & (R3 <= -6283184)) → nextState <= S3; CS = 0; else → nextState <= S1;	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	27'b0
S3	if(R3 > 141592 & (R3 <= 141592)) R4 ← R3 - 3.141592; R5 ← -1; if(R3 <= 141592 & (R3 <= -141592)) R4 ← R3 + 3.141592; R5 ← 1; else R4 ← R3; R5 ← 1;	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	27'bC01400
S4	if(R3 > 141592 & (R3 <= 141592)) R4 ← R3 - 3.141592; (Sub1) R5 ← 1; if(R3 <= 141592 & (R3 <= -141592)) R4 ← R3 + 3.141592; (Add1) R5 ← -1; else R4 ← R3; R5 ← 1;	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	27'bC01000
S5	R6 ← R1 x R5; (Mul1) R7 ← R2 x R5; (Mul2) R8 ← atanValue(numberOfIteration) x R5; (Mul3) R9 ← R6 numberOfIteration; (Sh1) R7 ← R7 numberOfIteration; (Sh2) R4 ← R4 - R8; (Sub2) R4 → 0; R1 ← R1 - R7; (Sub2) R2 ← R2 + R6; (Add1) R5 ← 1;	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	1	0	0	0	26'bC00000
S6	R4 ← 0; R1 ← R1 - R7; (Sub2) R2 ← R2 + R6; (Add1) R5 ← 1;	1	1	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	26'b6418000
S7	R9 ← R1 x scalingValue(numberOfIteration); (Mul1) R10 ← R2 x scalingValue(numberOfIteration); (Mul2) if(R3 > 141592 & (R3 <= 141592)) R9 ← R9 x (-1); (Neg1) R10 ← R10 x (-1); (Neg2)	0	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	26'b600C00
S8	if(R3 > 141592 & (R3 <= 141592)) R9 ← R9 x (-1); (Neg1) R10 ← R10 x (-1); (Neg2) else R9 ← R9; R10 ← R10;	0	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	0	26'b40000
	else R9 ← R9; R10 ← R10;	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	24'b0

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

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

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

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

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.

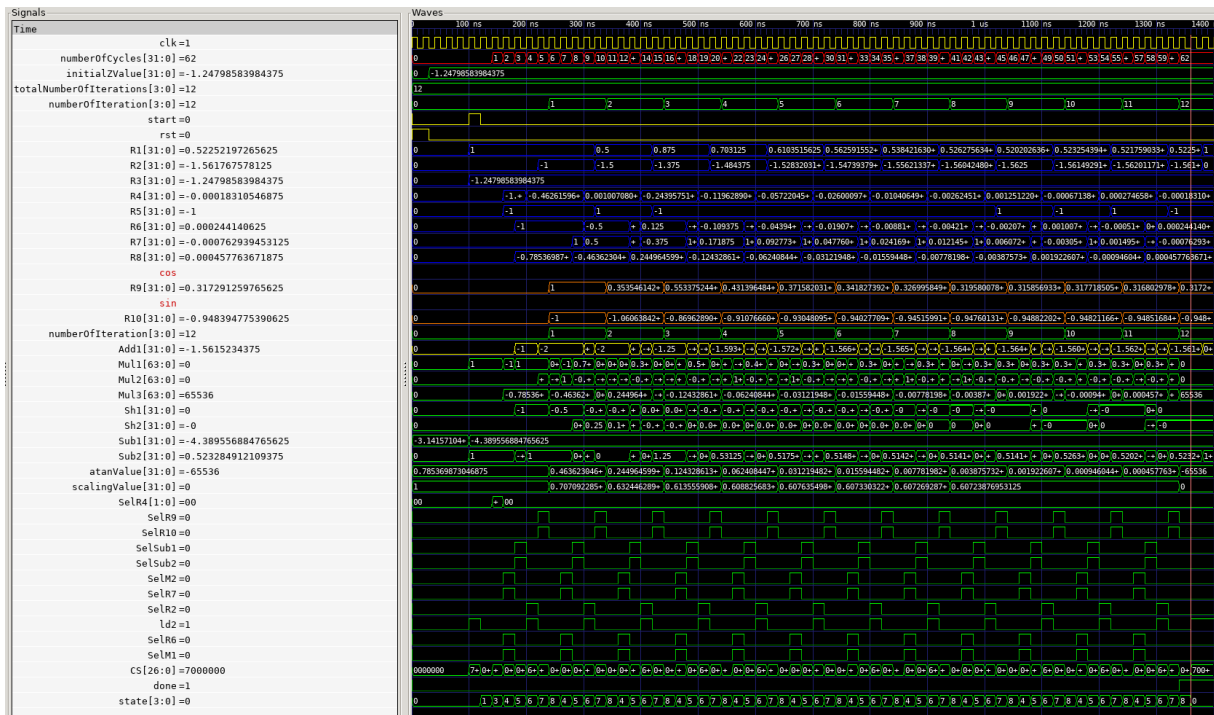


Figure 4 - 4 The whole Verilog simulation of CORDIC algorithm for determining the sinus and cosinus values of angle $\theta = -1.2479$ rad. The value of sinus and cosinus based on the current iteration is also calculated in this algorithm approach. The result is passed to the registers R9 and R10.

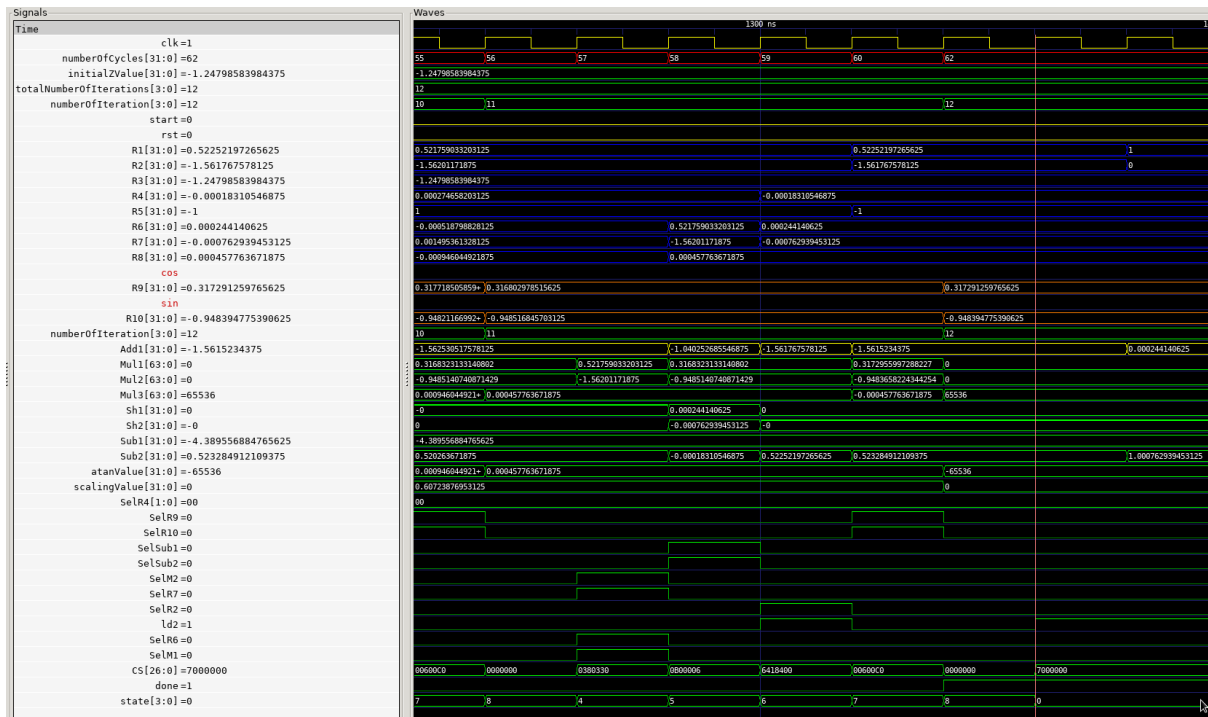


Figure 4 - 5 The detail of the last iteration of the Verilog simulation of CORDIC algorithm for determining the sinus and cosinus values of angle $\theta = -1.2479$ rad. The result is passed to the registers R9 and R10.

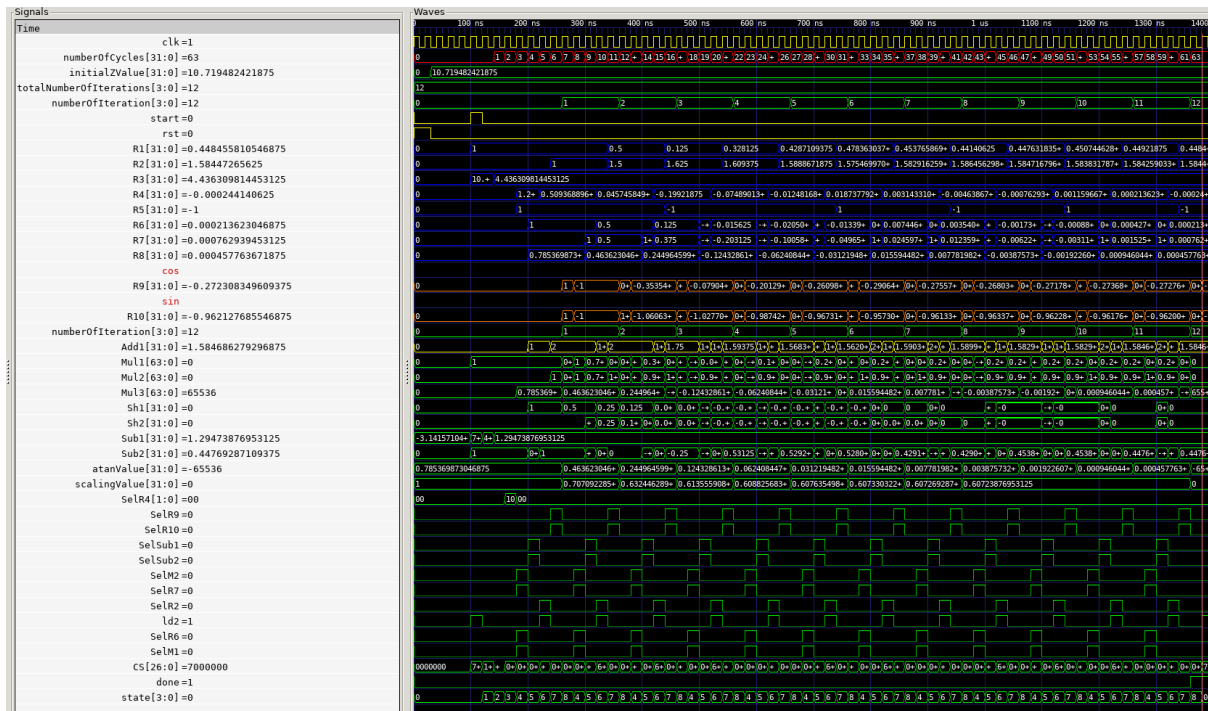


Figure 4 - 6 The whole Verilog simulation of CORDIC algorithm for determining the sinus and cosinus values of angle $\theta = 10.7195129$ rad. The value of sinus and cosinus based on the current iteration is also calculated in this algorithm approach. The result is passed to the registers R9 and R10.

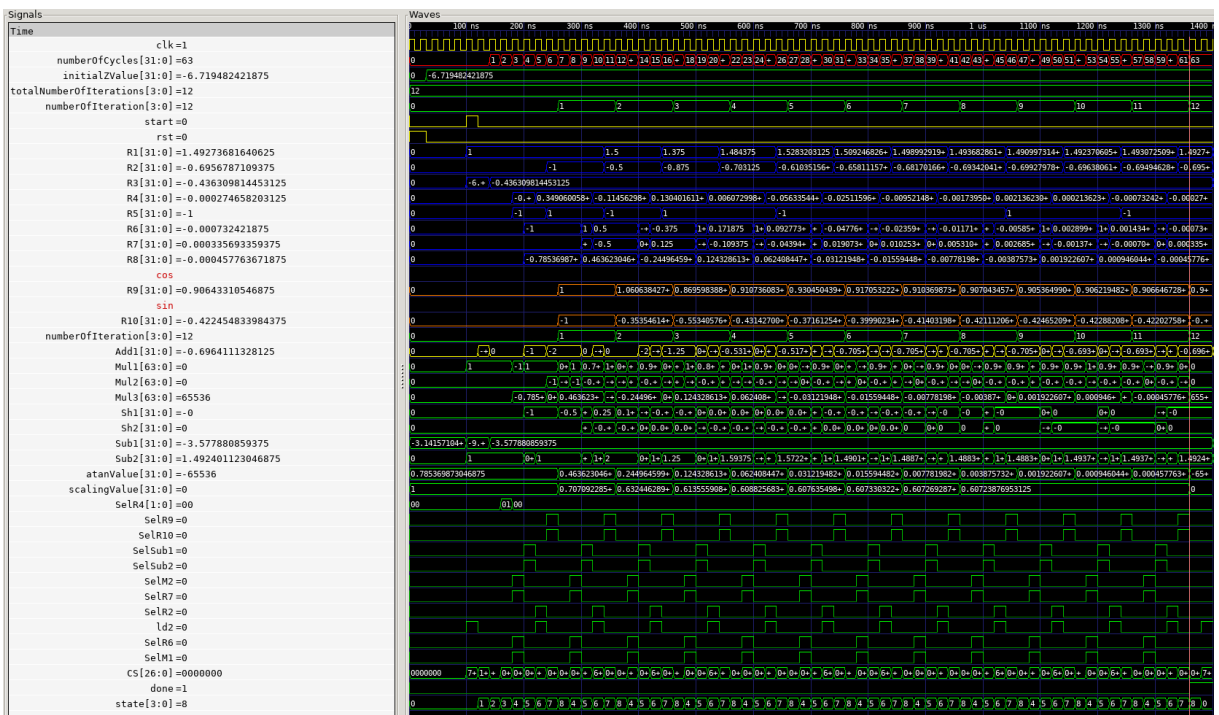


Figure 4 - 7 The whole Verilog simulation of CORDIC algorithm for determining the sinus and cosinus values of angle $\theta = -6.7195129$ rad. The value of sinus and cosinus based on the current iteration is also calculated in this algorithm approach. The result is passed to the registers R9 and R10.

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 Jacobi matrix and the defect.

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

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.

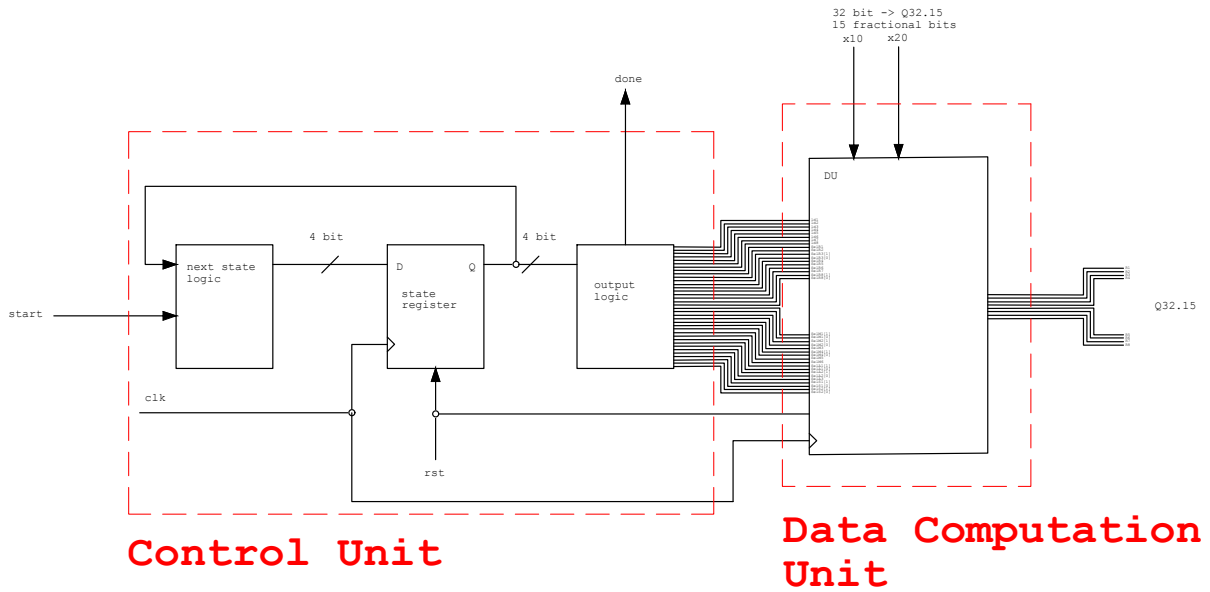


Figure 5 - 1 Top module design for the simple Newton-Raphson (NR) calculation unit module block design.

5.2.2 Allocation and Timing

The algorithm structure for the Verilog implementation is depicted in the data flow diagram in the picture 5 - 2. The algorithm iteration jumps (explained in the section *Control unit* of the simple NR algorithm) are not displayed in this diagram.

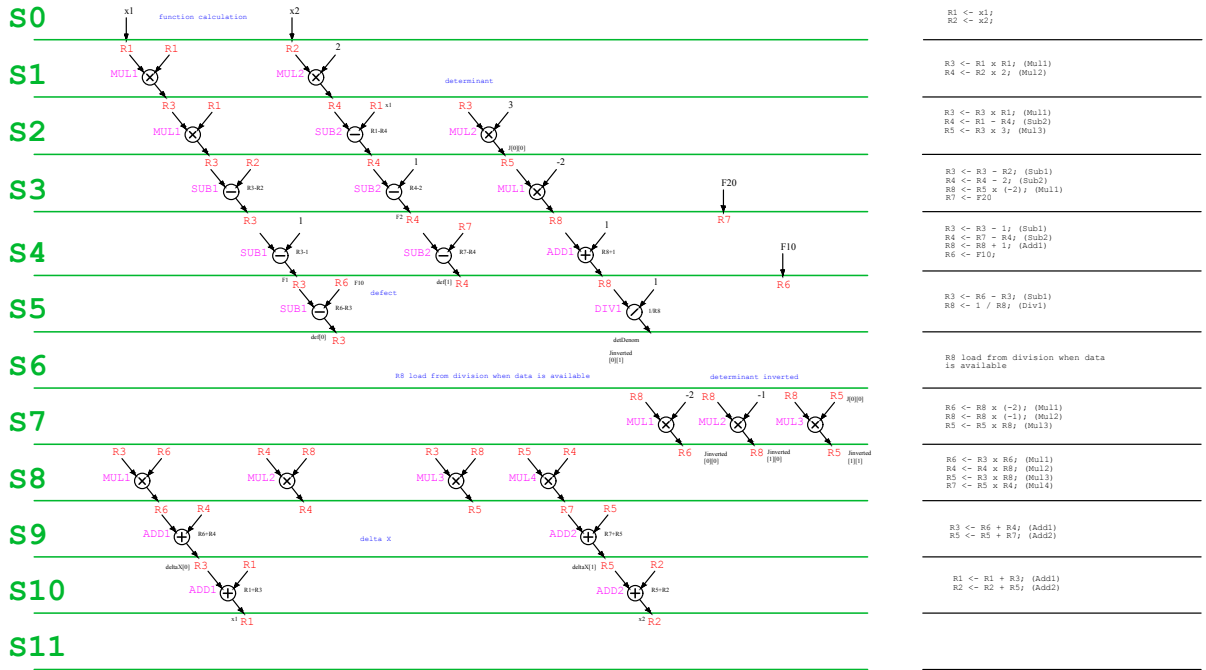


Figure 5 - 2 Allocation and timing diagram for the Data Path Unit part of the simple (NR) module.

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.

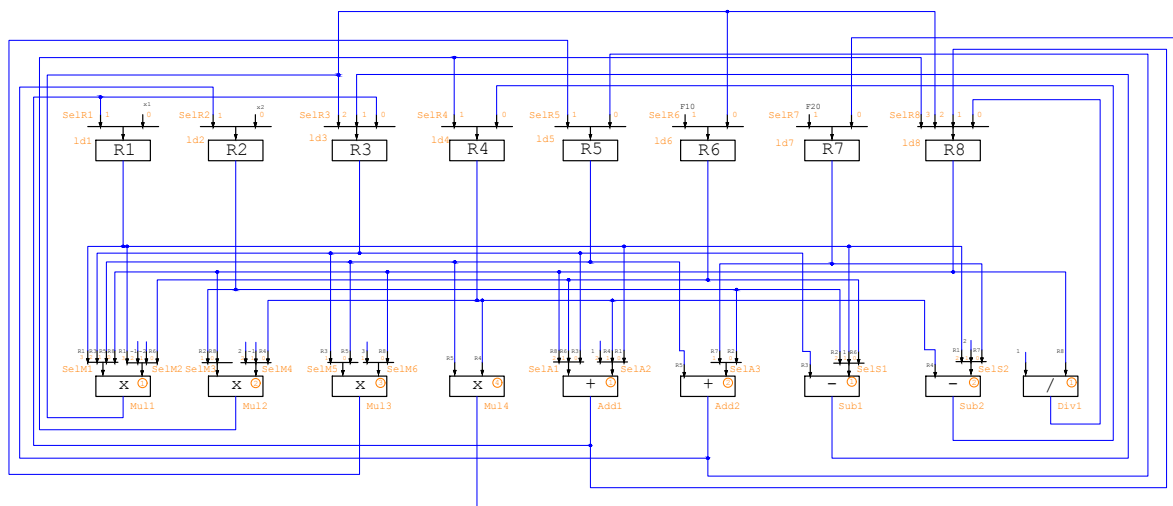


Figure 5 - 3 Register Transfer Level (RTL) scheme of the Data Path Unit part of the simple Newton-Raphson (NR) calculation IP.

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) alorithm solve Module.

State	RTL Code	35	34	33	32	31	30	29	28	27	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	CN
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	0	0	0	0	0	0	0	0	0	36'3C0000000	
S1	R3 ← R1 * R1 (1) R4 ← R2 * 2 (2)	0	0	1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	36'3B2B31000
S2	R5 ← R1 + R1 (1) R4 ← R1 - R4 (2) R5 ← R5 + 1 (3)	0	0	1	1	1	0	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	1	1	0	0	0	0	0	36'3B342C002
S3	R3 ← R3 - R2 (1) R4 ← R4 - 2 (2) R5 ← R5 + 2 (3) R7 ← R3	0	0	1	1	0	0	1	0	0	0	0	1	0	0	0	1	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	36'333119009
S4	R3 ← R3 - 1 (1) R4 ← R7 - R4 (2) R6 ← R4 + 1 (3) R8 ← R3	0	0	1	1	0	1	0	1	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	36'335124014	
S5	R5 ← R5 - R5 (1) R8 ← 1 / R5 (1)	0	0	1	0	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	0	0	0	0	0	36'323100000	
S6	R6 load from memory when data is available	0	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	0	0	0	0	0	0	0	0	36'310000000	
S7	R6 ← R6 + 1 (1) R5 ← R5 + 1 (2) R5 ← R5 - R6 (3)	0	0	0	0	1	1	0	1	0	0	0	0	0	0	1	0	0	1	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	36'3404C4000
S8	R3 ← 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	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	36'3410C2000	
S9	R3 ← R4 + R4 (1) R5 ← R5 + R5 (2)	0	0	1	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	1	0	1	1	0	0	0	36'320000000	
S10	R1 ← R1 - R3 (1) R2 ← R2 - R5 (2)	1	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	36'3C3C00000	
S11		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	36'000000000		

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.

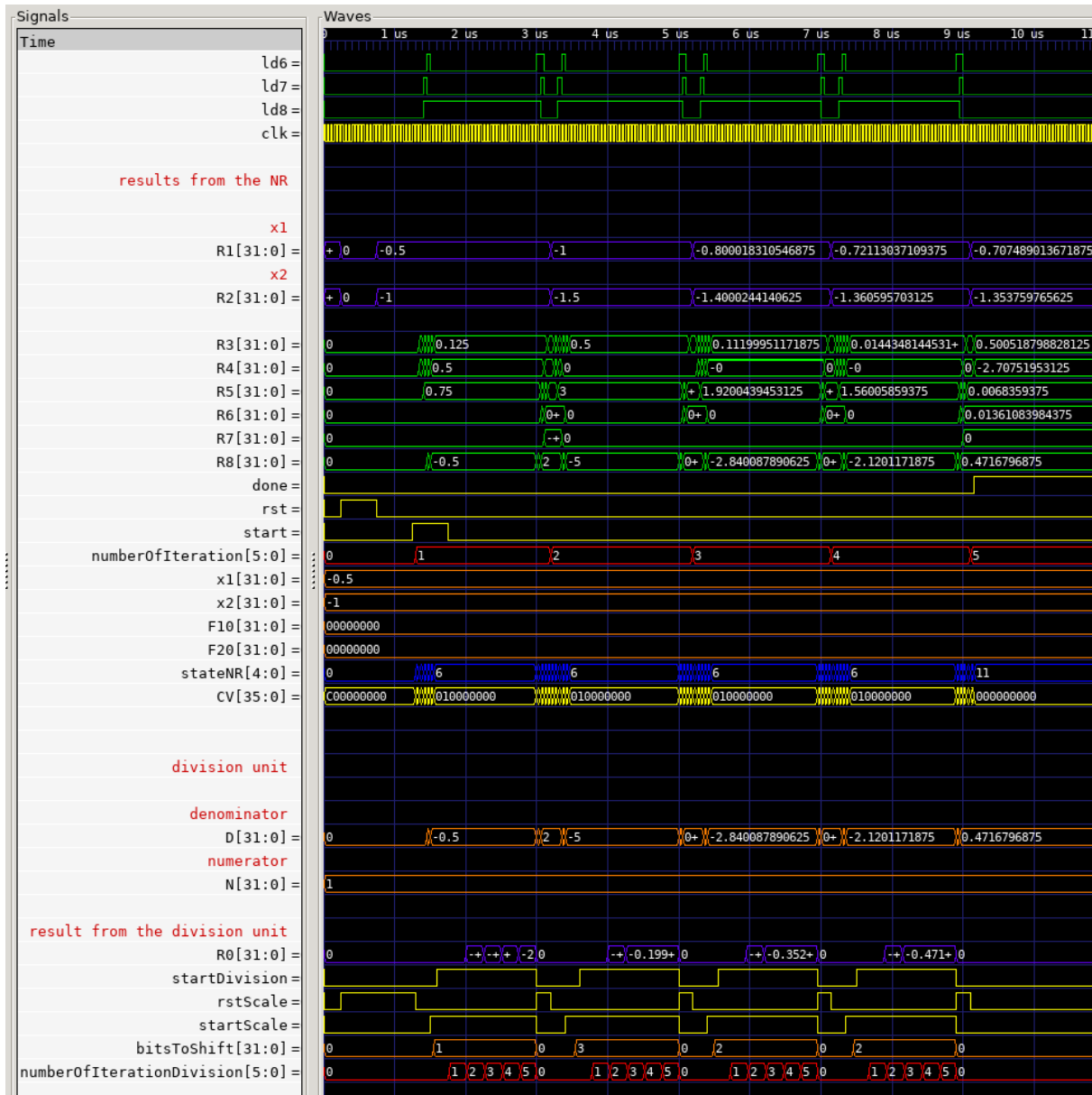


Figure 5 - 4 The whole Verilog simulation of a simple Newton-Raphson (NR) algorithm. The result is may be seen in registers R1 and R2 after the fifth iteration of the algorithm.

6 Selective Harmonic Elimination

6.1 Theory

6.2 IP Block Design

6.2.1 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 ??.

6.2.2 Allocation and Timing

The Allocation and Timing diagram, depicted on Figure ?? 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.

6.2.3 Data Path Unit

6.2.4 Control Unit

Table 6 - 1 Control signal encoding table for instructions to be processed by the simple Newton-Raphson (NR) alogrithm solve Module.

Inst	Op	Op2	Op3	Op4	Op5	Op6	Op7	Op8	Op9	Op10	Op11	Op12	Op13	Op14	Op15	Op16	Op17	Op18	Op19	Op20	Op21	Op22	Op23	Op24	Op25	Op26	Op27	Op28	Op29	Op30	Op31	Op32	Op33	Op34	Op35	Op36	Op37	Op38	Op39	Op40	Op41	Op42	Op43	Op44	Op45	Op46	Op47	Op48	Op49	Op50	Op51	Op52	Op53	Op54	Op55	Op56	Op57	Op58	Op59	Op60	Op61	Op62	Op63	Op64	Op65	Op66	Op67	Op68	Op69	Op70	Op71	Op72	Op73	Op74	Op75	Op76	Op77	Op78	Op79	Op80	Op81	Op82	Op83	Op84	Op85	Op86	Op87	Op88	Op89	Op90	Op91	Op92	Op93	Op94	Op95	Op96	Op97	Op98	Op99	Op100	Op101	Op102	Op103	Op104	Op105	Op106	Op107	Op108	Op109	Op110	Op111	Op112	Op113	Op114	Op115	Op116	Op117	Op118	Op119	Op120	Op121	Op122	Op123	Op124	Op125	Op126	Op127	Op128	Op129	Op130	Op131	Op132	Op133	Op134	Op135	Op136	Op137	Op138	Op139	Op140	Op141	Op142	Op143	Op144	Op145	Op146	Op147	Op148	Op149	Op150	Op151	Op152	Op153	Op154	Op155	Op156	Op157	Op158	Op159	Op160	Op161	Op162	Op163	Op164	Op165	Op166	Op167	Op168	Op169	Op170	Op171	Op172	Op173	Op174	Op175	Op176	Op177	Op178	Op179	Op180	Op181	Op182	Op183	Op184	Op185	Op186	Op187	Op188	Op189	Op190	Op191	Op192	Op193	Op194	Op195	Op196	Op197	Op198	Op199	Op200	Op201	Op202	Op203	Op204	Op205	Op206	Op207	Op208	Op209	Op210	Op211	Op212	Op213	Op214	Op215	Op216	Op217	Op218	Op219	Op220	Op221	Op222	Op223	Op224	Op225	Op226	Op227	Op228	Op229	Op230	Op231	Op232	Op233	Op234	Op235	Op236	Op237	Op238	Op239	Op240	Op241	Op242	Op243	Op244	Op245	Op246	Op247	Op248	Op249	Op250	Op251	Op252	Op253	Op254	Op255	Op256	Op257	Op258	Op259	Op260	Op261	Op262	Op263	Op264	Op265	Op266	Op267	Op268	Op269	Op270	Op271	Op272	Op273	Op274	Op275	Op276	Op277	Op278	Op279	Op280	Op281	Op282	Op283	Op284	Op285	Op286	Op287	Op288	Op289	Op290	Op291	Op292	Op293	Op294	Op295	Op296	Op297	Op298	Op299	Op300	Op301	Op302	Op303	Op304	Op305	Op306	Op307	Op308	Op309	Op310	Op311	Op312	Op313	Op314	Op315	Op316	Op317	Op318	Op319	Op320	Op321	Op322	Op323	Op324	Op325	Op326	Op327	Op328	Op329	Op330	Op331	Op332	Op333	Op334	Op335	Op336	Op337	Op338	Op339	Op340	Op341	Op342	Op343	Op344	Op345	Op346	Op347	Op348	Op349	Op350	Op351	Op352	Op353	Op354	Op355	Op356	Op357	Op358	Op359	Op360	Op361	Op362	Op363	Op364	Op365	Op366	Op367	Op368	Op369	Op370	Op371	Op372	Op373	Op374	Op375	Op376	Op377	Op378	Op379	Op380	Op381	Op382	Op383	Op384	Op385	Op386	Op387	Op388	Op389	Op390	Op391	Op392	Op393	Op394	Op395	Op396	Op397	Op398	Op399	Op400	Op401	Op402	Op403	Op404	Op405	Op406	Op407	Op408	Op409	Op410	Op411	Op412	Op413	Op414	Op415	Op416	Op417	Op418	Op419	Op420	Op421	Op422	Op423	Op424	Op425	Op426	Op427	Op428	Op429	Op430	Op431	Op432	Op433	Op434	Op435	Op436	Op437	Op438	Op439	Op440	Op441	Op442	Op443	Op444	Op445	Op446	Op447	Op448	Op449	Op450	Op451	Op452	Op453	Op454	Op455	Op456	Op457	Op458	Op459	Op460	Op461	Op462	Op463	Op464	Op465	Op466	Op467	Op468	Op469	Op470	Op471	Op472	Op473	Op474	Op475	Op476	Op477	Op478	Op479	Op480	Op481	Op482	Op483	Op484	Op485	Op486	Op487	Op488	Op489	Op490	Op491	Op492	Op493	Op494	Op495	Op496	Op497	Op498	Op499	Op500	Op501	Op502	Op503	Op504	Op505	Op506	Op507	Op508	Op509	Op510	Op511	Op512	Op513	Op514	Op515	Op516	Op517	Op518	Op519	Op520	Op521	Op522	Op523	Op524	Op525	Op526	Op527	Op528	Op529	Op530	Op531	Op532	Op533	Op534	Op535	Op536	Op537	Op538	Op539	Op540	Op541	Op542	Op543	Op544	Op545	Op546	Op547	Op548	Op549	Op550	Op551	Op552	Op553	Op554	Op555	Op556	Op557	Op558	Op559	Op560	Op561	Op562	Op563	Op564	Op565	Op566	Op567	Op568	Op569	Op570	Op571	Op572	Op573	Op574	Op575	Op576	Op577	Op578	Op579	Op580	Op581	Op582	Op583	Op584	Op585	Op586	Op587	Op588	Op589	Op590	Op591	Op592	Op593	Op594	Op595	Op596	Op597	Op598	Op599	Op600	Op601	Op602	Op603	Op604	Op605	Op606	Op607	Op608	Op609	Op610	Op611	Op612	Op613	Op614	Op615	Op616	Op617	Op618	Op619	Op620	Op621	Op622	Op623	Op624	Op625	Op626	Op627	Op628	Op629	Op630	Op631	Op632	Op633	Op634	Op635	Op636	Op637	Op638	Op639	Op640	Op641	Op642	Op643	Op644	Op645	Op646	Op647	Op648	Op649	Op650	Op651	Op652	Op653	Op654	Op655	Op656	Op657	Op658	Op659	Op660	Op661	Op662	Op663	Op664	Op665	Op666	Op667	Op668	Op669	Op670	Op671	Op672	Op673	Op674	Op675	Op676	Op677	Op678	Op679	Op680	Op681	Op682	Op683	Op684	Op685	Op686	Op687	Op688	Op689	Op690	Op691	Op692	Op693	Op694	Op695	Op696	Op697	Op698	Op699	Op700	Op701	Op702	Op703	Op704	Op705	Op706	Op707	Op708	Op709	Op710	Op711	Op712	Op713	Op714	Op715	Op716	Op717	Op718	Op719	Op720	Op721	Op722	Op723	Op724	Op725	Op726	Op727	Op728	Op729	Op730	Op731	Op732	Op733	Op734	Op735	Op736	Op737	Op738	Op739	Op740	Op741	Op742	Op743	Op744	Op745	Op746	Op747	Op748	Op749	Op750	Op751	Op752	Op753	Op754	Op755	Op756	Op757	Op758	Op759	Op760	Op761	Op762	Op763	Op764	Op765	Op766	Op767	Op768	Op769	Op770	Op771	Op772	Op773	Op774	Op775	Op776	Op777	Op778	Op779	Op780	Op781	Op782	Op783	Op784	Op785	Op786	Op787	Op788	Op789	Op790	Op791	Op792	Op793	Op794	Op795	Op796	Op797	Op798	Op799	Op800	Op801	Op802	Op803	Op804	Op805	Op806	Op807	Op808	Op809	Op810	Op811	Op812	Op813	Op814	Op815	Op816	Op817	Op818	Op819	Op820	Op821	Op822	Op823	Op824	Op825	Op826	Op827	Op828	Op829	Op830	Op831	Op832	Op833	Op834	Op835	Op836	Op837	Op838	Op839	Op840	Op841	Op842	Op843	Op844	Op845	Op846	Op847	Op848	Op849	Op850	Op851	Op852	Op853	Op854	Op855	Op856	Op857	Op858	Op859	Op860	Op861	Op862	Op863	Op864	Op865	Op866	Op867	Op868	Op869	Op870	Op871	Op872	Op873	Op874	Op875	Op876	Op877	Op878	Op879	Op880	Op881	Op882	Op883	Op884	Op885	Op886	Op887	Op888	Op889	Op890	Op891	Op892	Op893	Op894	Op895	Op896	Op897	Op898	Op899	Op900	Op901	Op902	Op903	Op904	Op905	Op906	Op907	Op908	Op909	Op910	Op911	Op912	Op913	Op914	Op915	Op916	Op917	Op918	Op919	Op920	Op921	Op922	Op923	Op924	Op925	Op926	Op927	Op928	Op929	Op930	Op931	Op932	Op933	Op934	Op935	Op936	Op937	Op938	Op939	Op940	Op941	Op942	Op943	Op944	Op945	Op946	Op947	Op948	Op949	Op950	Op951	Op952	Op953	Op954	Op955	Op956	Op957	Op958	Op959	Op960	Op961	Op962	Op963	Op964	Op965	Op966	Op967	Op968	Op969	Op970	Op971	Op972	Op973	Op974	Op975	Op976	Op977	Op978	Op979	Op980	Op981	Op982	Op983	Op984	Op985	Op986	Op987	Op988	Op989	Op990	Op991	Op992	Op993	Op994	Op995	Op996	Op997	Op998	Op999	Op1000
00	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	00																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																															

Conclusion

And this is the conclusion of my report. P_n .

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] 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.
- [4] MEYER-BÄSE, Uwe. *Digital signal processing with field programmable gate arrays*. 4th ed. Berlin: Springer, 2014. ISBN 978-3-642-45308-3.
- [5] 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.
- [6] 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.
- [7] 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 Symbols and Abbreviations

A.1 List of abbreviations

CORDIC	Coordinate Rotation Digital Computer
CPU	Central Processing Unit
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

A.2 List of symbols

P_n	(W)	jmenovitý výkon stroje
-------	-----	------------------------