CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering

Department of Electric Drives and Traction

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

Technical report

Petr Zakopal

Prague 2023

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction

This 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*.

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

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.

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

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

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

Because finding root of the equation 3 - 2, the value of $F(x_{i+1})$ is set to be zero. After separating the x_{i+1} value of the 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].

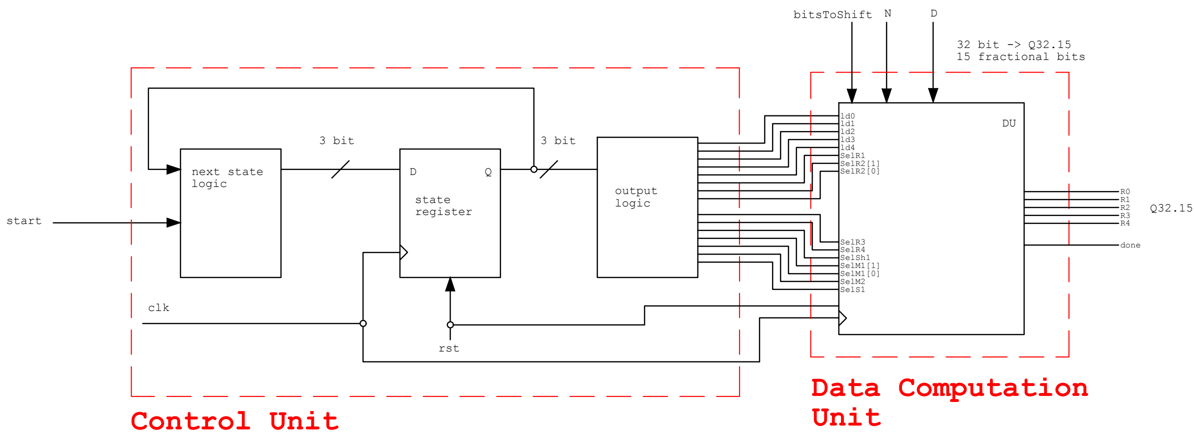

Figure 3 - 1 Top module design for the division unit module block design.

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.

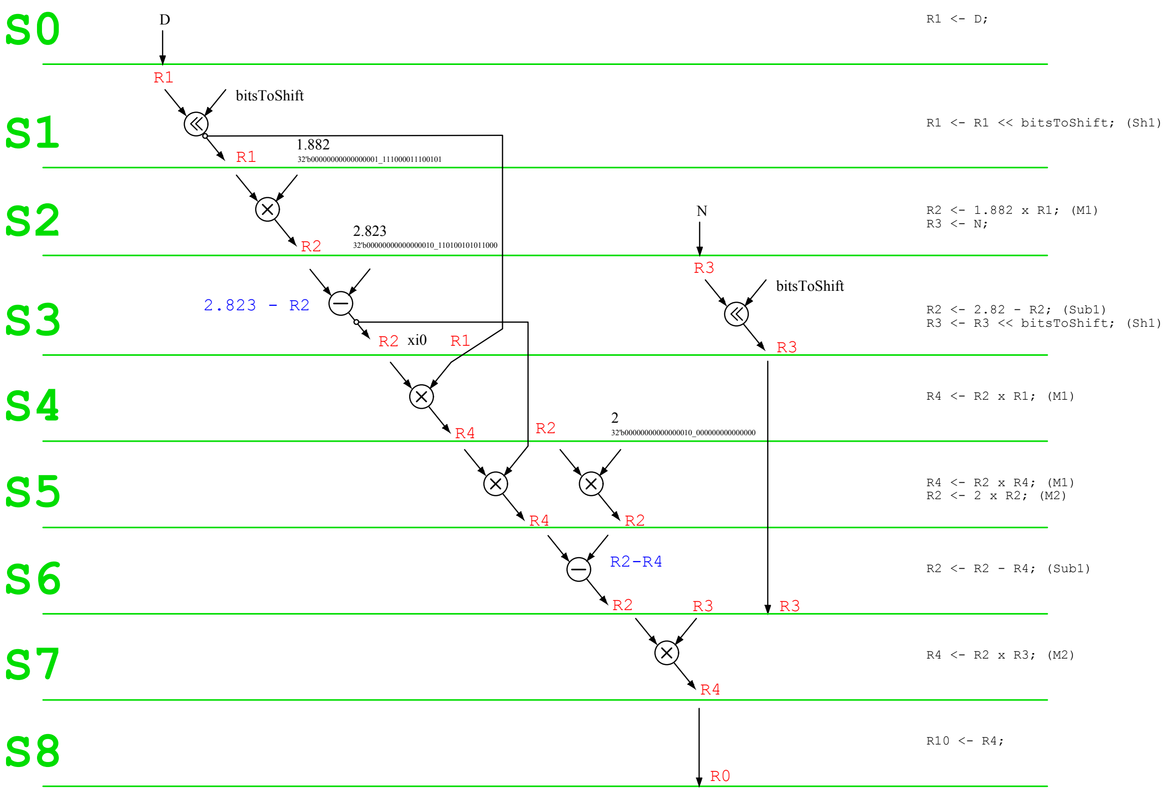

Figure 3 - 2 Alloccation and timing diagram for the Data Path Unit part of the division module.

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.

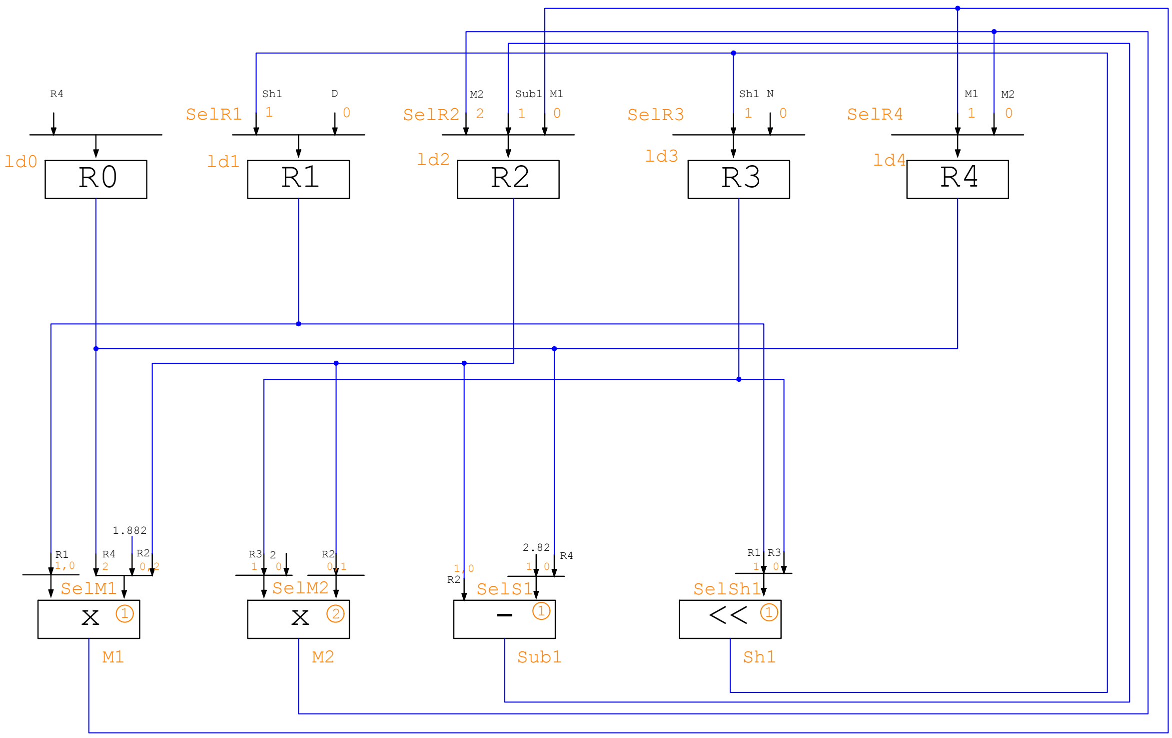

Figure 3 - 3 Register Transfer Level (RTL) scheme of the Data Path Unit part of the division module.

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

3.3 Calculating number of bits to shift the denominator

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

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

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

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

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

```
 timescale 1ns/1ns
 #10; // wait for 10 units of time
 #0 rstScale = 1; startScale = 0; // reset unit for determining the number of bits to shift in the denominator and do not start the unit yet
 N = 32'b00000000100110000_000010000000000; D=32'b1111111111111111_110000000000000; // set the numerator to N = 304.03125, denominator to D = -0.25
 #10 rstScale = 0; // wait for 10 units of time and stop the reset of scaling unit
 #10 startScale = 1; // start the algorithm for scaling unit
 #20 rst = 1; start = 0; // reset the division unit
 #30 rst = 0; // stop reseting of the division unit
 #20 start = 1; // start the division unit
 #20 start = 0;
 #1000; // wait 1000 units of time
 $finish; // finish the simulation
```

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

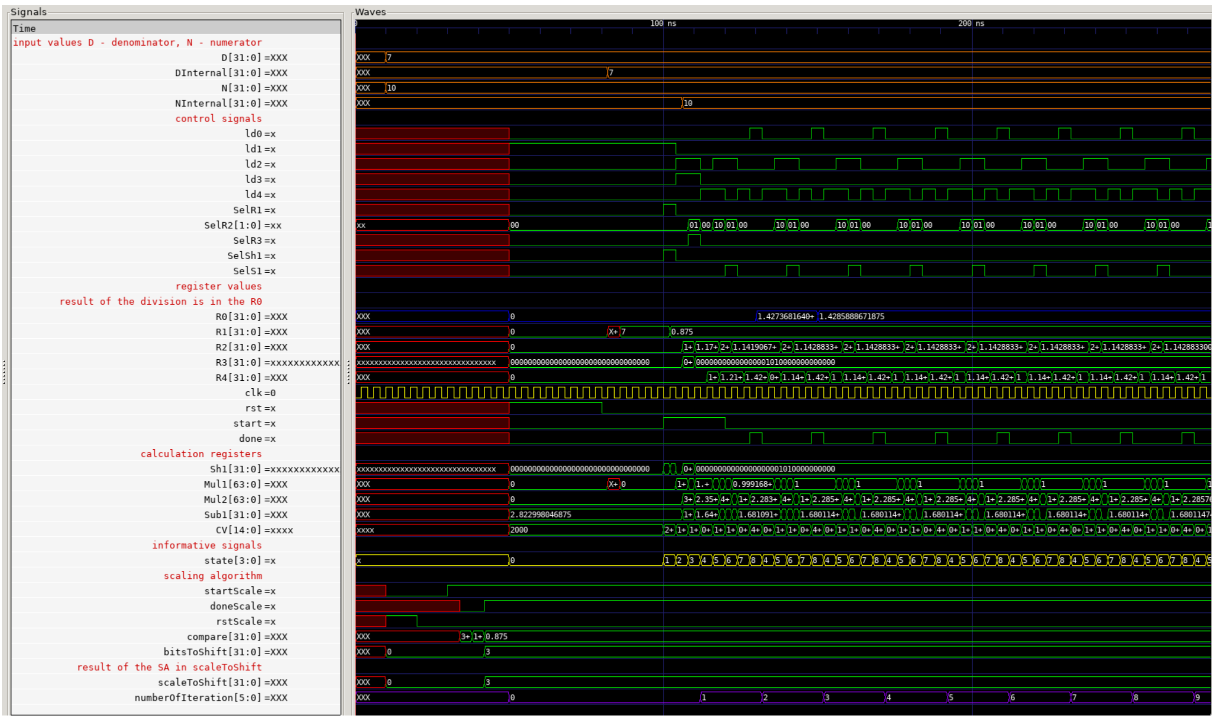

Figure 3 - 4 Selected signals of simulation of division N/D = 10 / 7. The correct result in R0 is obtained after two iterations (reg numberOfIterations).

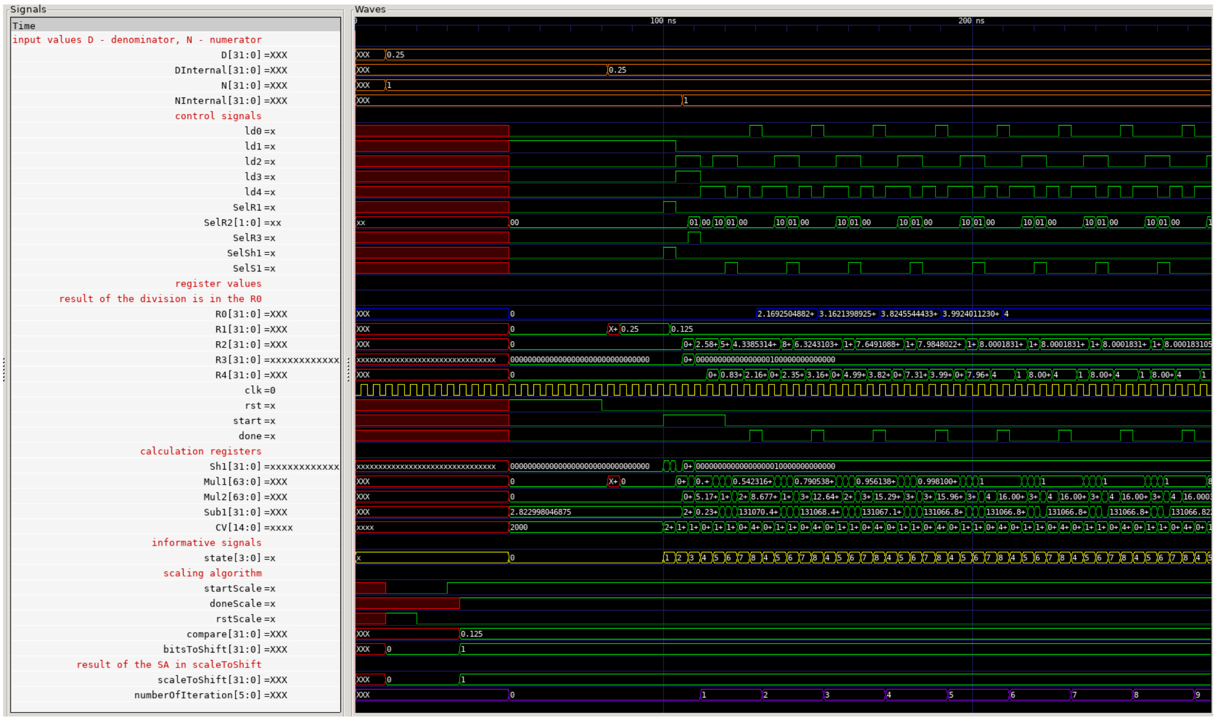

Figure 3 - 5 Selected signals of simulation of division N/D = 1 / 0.25. The correct result in R0 is obtained after five iterations (reg numberOfIterations).

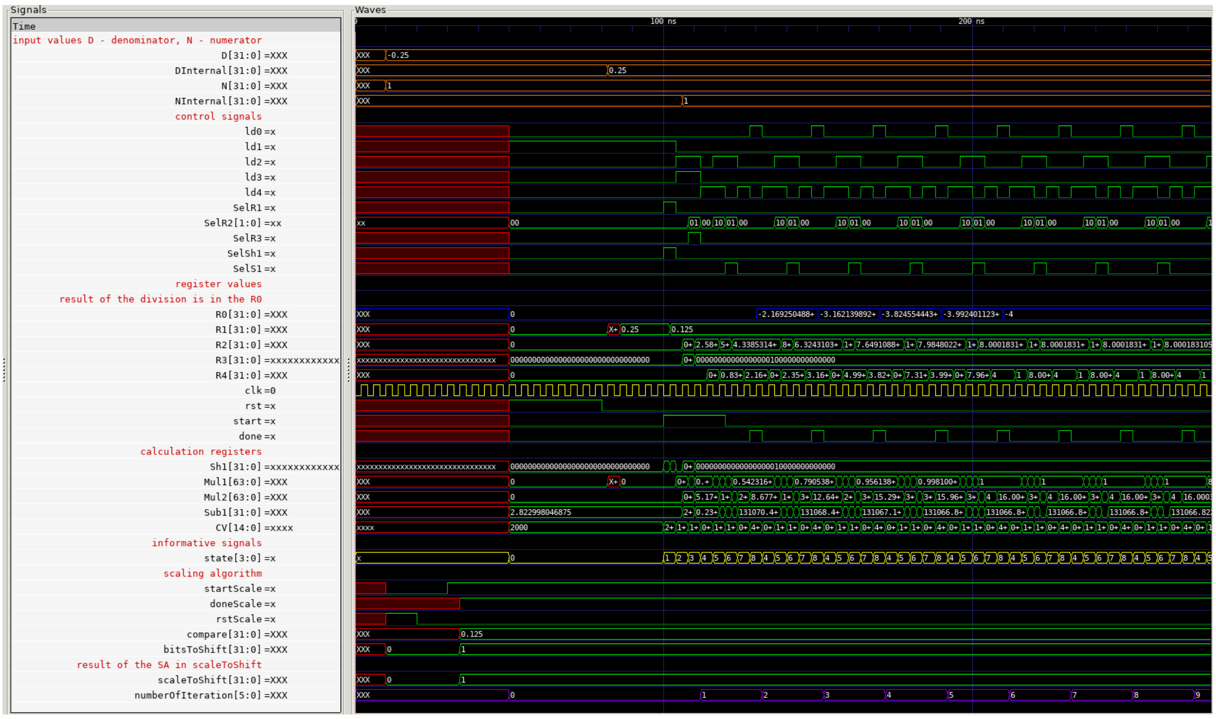

Figure 3 - 6 Selected signals of simulation of division N/D = 1 / (-0.25). The correct result in R0 is obtained after five iterations (reg numberOfIterations).

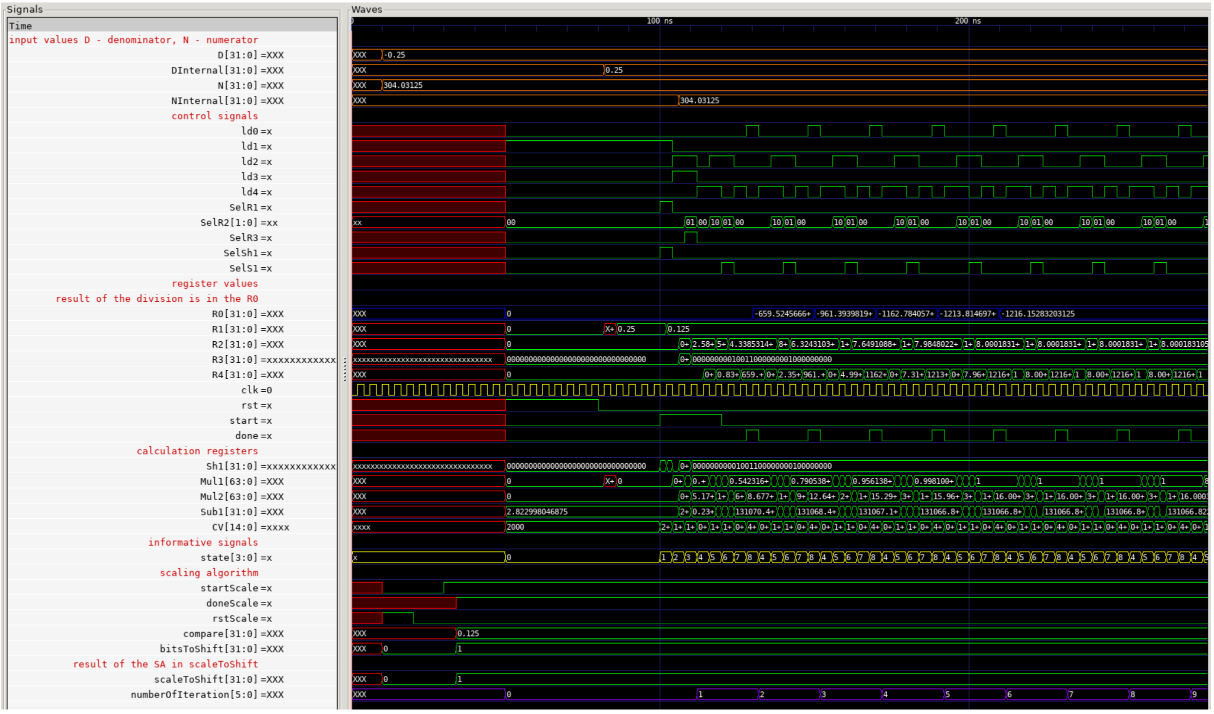

Figure 3 - 7 Selected signals of simulation of division N/D = 304.03215 / (-0.25). The correct result in R0 is obtained after five iterations (reg numberOfIterations).

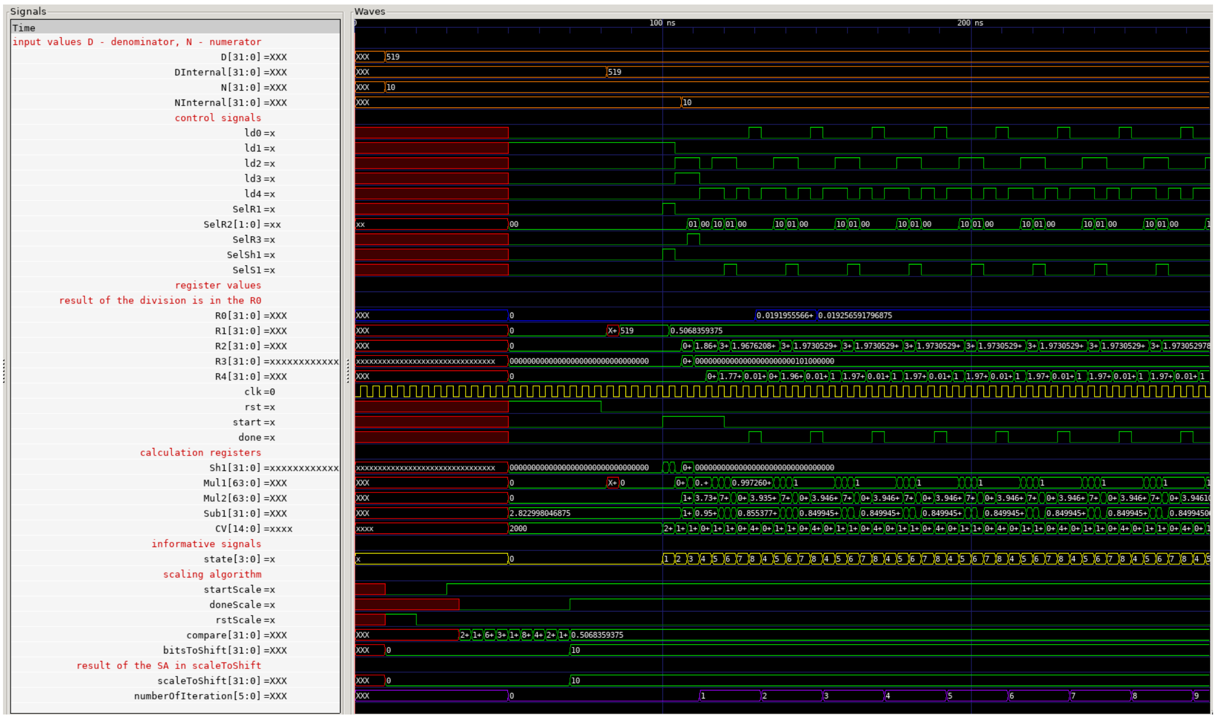

Figure 3 - 8 Selected signals of simulation of division N/D = 10 / (519). The correct result in R0 is obtained after two iterations (reg numberOfIterations).

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.

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

where x_{R} and y_{R} are coordinates of a rotated vector, θ is the angle for which the vector with coordinates x_{in} and y_{in} was rotated.

Then when simplifying the equation

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

it can be seen, that only multiplication by scaling factor of precalculated values of $\cos(\theta)$, multiplication by $\tan(\theta)$, subtraction and addiction operations are needed. 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.

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

When the values $x_{\mathrm{in}} = 1$ and $y_{\mathrm{in}} = 0$ 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.

$$\begin{aligned} x_{\mathrm{R}} &= x_{\mathrm{in}} \cos(\theta) - y_{\mathrm{in}} \sin(\theta) = |\theta = 0| = \cos(\theta) \\ y_{\mathrm{R}} &= x_{\mathrm{in}} \sin(\theta) + y_{\mathrm{in}} \cos(\theta) = |\theta = 0| = \sin(\theta) \end{aligned} \tag{4 - 4}$$

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 multipliying $cosinus$ of different θ values converges to $0,60725$. So there is no need to precalculate all the scaling values only the convergenent 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.

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

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

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

The algorithm as presented 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 transfered 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 a 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.

```
if((initialZValue > 1.5707)&(initialZValue < 3.141592))
    sigmaValue = -1
    zValue = initialZValue - 3.141592
else if((initialZValue > 3.141592)&(initialZValue < 4.7123))
    sigmaValue = 1
    zValue = initialZValue - 3.141592
else
    zValue = initialZValue
    sigmaValue = 1
end
```

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

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

```
    resultCos = - resultCos
    resultSin = resultSin
else if((initialZValue > 3.141592)&(initialZValue < 4.7123))
    resultCos = - resultCos
    resultSin = - resultSin
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.

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

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

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

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

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

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.

```
import math
```

```python

# Defining starting values and empty arrays
totalNumberOfIterations = 12 # 12 - best tradeof between value and iterations
atanValues = []
scalingValues = [1]
initialXValueCordic = 1
initialYValueCordic = 0
# initialZValueCordic = 1.248  # angle for which to calculate cordic
# initialZValueCordic = - 1.248  # angle for which to calculate cordic
# initialZValueCordic = - 6.7194  # angle for which to calculate cordic
initialZValueCordic = 10.7194824  # angle for which to calculate cordic
initialSigmaValueCordic = 1

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

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

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

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


print("initialZValue after moving to [0,2Pi] interval: ", initialZValueCordic)
print("\n")
print("*-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-*")

# Checking the initial value and moving it in the interval
if (initialZValueCordic > 1.5707) and (initialZValueCordic < 3.141592):
    zValue = initialZValueCordic - 3.141592
    sigmaValue = -1
    print("value in second q")
```

```python
elif (initialZValueCordic > 3.141592) and (initialZValueCordic < 4.7123):
    zValue = initialZValueCordic - 3.141592
    sigmaValue = 1
    print("value in third q")
elif (initialZValueCordic < 0):
    sigmaValue = -1
    zValue = initialZValueCordic
    print("value in fourth q")
else:
    zValue = initialZValueCordic  # For angle
    sigmaValue = initialSigmaValueCordic  # For +- next angle
    print("value in first")

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


# CORDIC ALGORITHM
for x in range(totalNumberOfIterations):

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

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

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

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

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

# Changing results sign based on the rotation of the initialZValueCordic
if (initialZValueCordic > 1.5707) and (initialZValueCordic < 3.141592):
```

```python
    resultCos = - resultCos
elif (initialZValueCordic > 3.141592) and (initialZValueCordic < 4.7123):
    resultCos = - resultCos
    resultSin = - resultSin

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

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

# Results printing
print("*-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-*")
print("CORDIC results:")
print("cos: ", resultCos)
print("sin: ", resultSin)
print("scaleFactor: ", scalingValues[totalNumberOfIterations-1])
print("\n")
print("MATH results:")
print("cos: ", mathResultCos)
print("sin: ", mathResultSin)
print("\n")
print("error CORDIC-MATH:")
print("cos: ", errorCos)
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. Obivously the already presented custom fixed point

Q32.15 format is required.

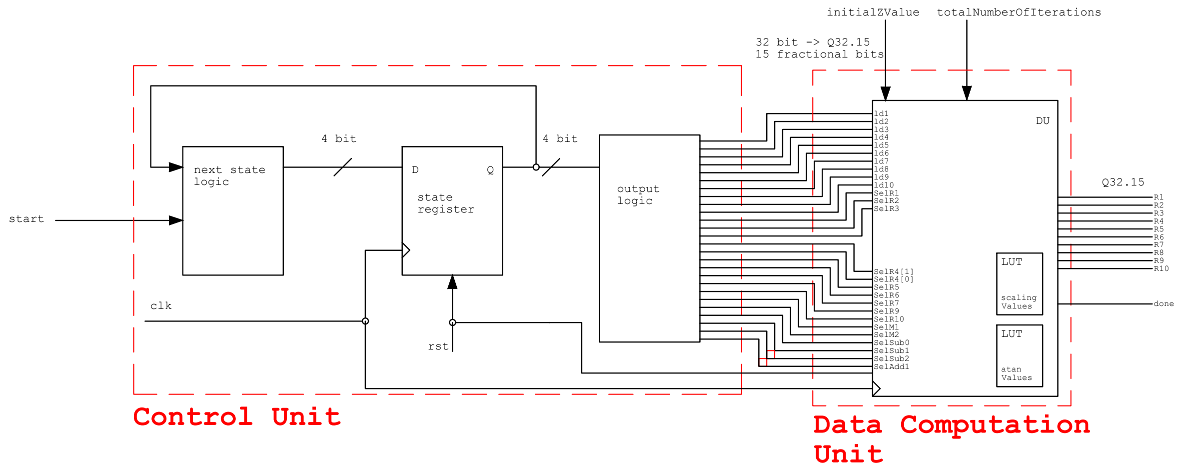

Figure 4 - 1 Top module design for the CORDIC module block design.

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 aproach. The jumps in the allocation and timing diagram are not shown.

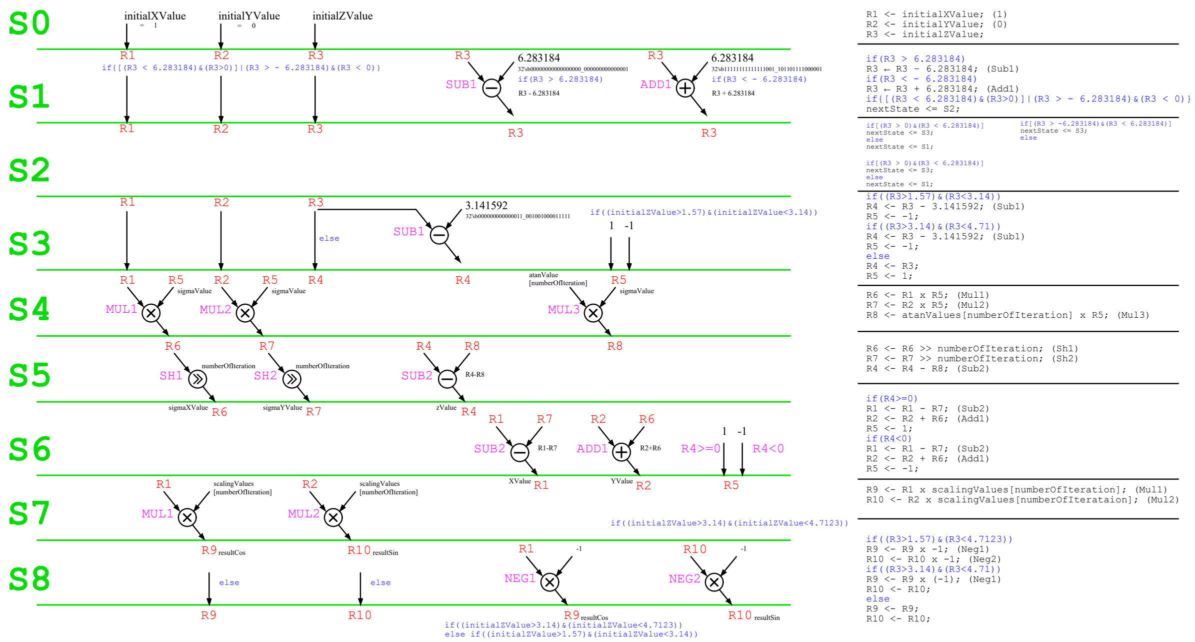

Figure 4 - 2 Alloccation and timing diagram for the Data Path Unit part of the CORDIC IP.

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 $\textbf{SelR}_x$ 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.

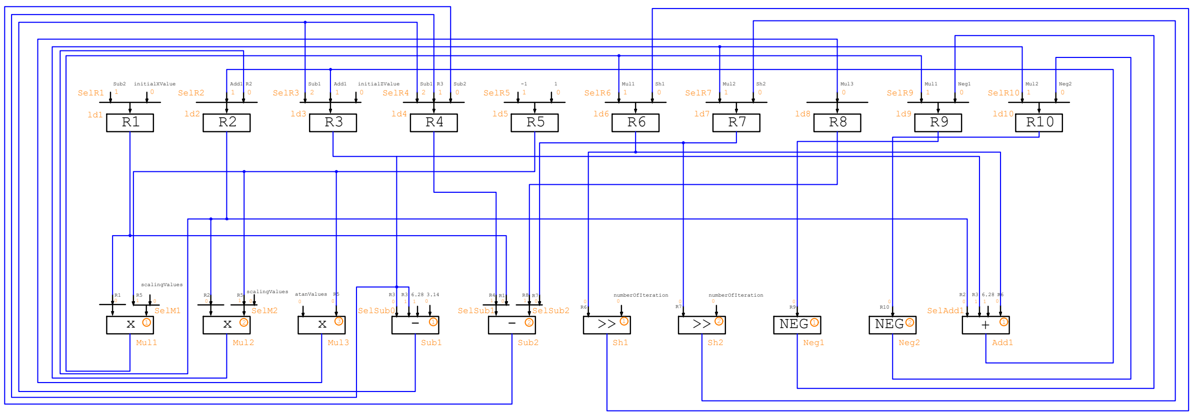

Figure 4 - 3 Register transfer level RTL scheme of the CORDIC IP Data Path Unit IP.

```
module atanValuesCordicLUT(index, returnValue);

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


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

```
        4'b0111: returnValue = 32'sb00000000000000000_000000011111111; //
    0.007812341060101111
        4'b1000: returnValue = 32'sb00000000000000000_000000001111111; //
    0.007812341060101111
        4'b1001: returnValue = 32'sb00000000000000000_000000000111111; //
    0.0019531225164788188
        4'b1010: returnValue = 32'sb00000000000000000_000000000011111; //
    0.0009765621895593195
        4'b1011: returnValue = 32'sb00000000000000000_000000000001111; //
    0.0004882812111948983
        default: returnValue = 32'sb00000000000000000_000000000000000; // 0
    endcase
end
endmodule
```

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

```
module scalingValuesCordicLUT(index, returnValue);

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

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

```
        default: returnValue <= 32'sb00000000000000000_000000000000000; // 0
    endcase
end
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 iteation 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 ld1	25 ld2	24 ld3	23 ld4	22 ld5	21 ld6	20 ld7	19 ld8	18 ld9	17 ld10	16 SelR1	15 SelR2	14 SelR3[1]	13 SelR3[0]	12 SelR4[1]	11 SelR4[0]	10 SelR5	9 SelR6	8 SelR7	7 SelR9	6 SelR10	5 SelM1	4 SelM2	3 SelSub0	2 SelSub1	1 SelSub2	0 SelAdd1	CS
S0	R0 ← totalNumberOfIterations; R1 ← initialXValue; R2 ← initialYValue; R3 ← initialZValue;	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	27'h7000000
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	27'h1004008
	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	27'h1002001
	if{[(R3 <6.283184)&(R3>0)]|(R3 >- 6.283184)&(R3 <0)} → nextState <= S2;																												27'h0
S2	if[(R3 >0)&(R3 <6.283184)] → nextState <= S3; CS = 0; else → nextState <= S1; if[(R3 >0)&(R3 <6.283184)] → nextState <= S3; CS = 0; else → nextState <= S1; if[(R3 >-6.283184)&(R3 <6.283184)] → nextState <= S3; CS = 0; else → nextState <= S1;	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
S3	if[(R3 >1.5707)&(R3 <3.141592)] R4 ← R3 – 3.141592; (Sub1) R5 ← -1;	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	27'hC01400
	if[(R3 >3.141592)&(R3 <4.7123)] R4 ← R3 – 3.141592; (Sub1) R5 ← 1;	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	27'hC01000
	if(R3 <0) R5 ← -1; R4 ← R3;	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	27'hC00C00
	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	26'hC00800
S4	R6 ← R1 x R5; (Mul1) R7 ← R2 x R5; (Mul2) R8 ← atanValues[numberOfIteration] x R5; (Mul3)	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	0	0	0	0	26'h380330
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	26'hB00006
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	26'h6418000
	R4 <0; R1 ← R1 – R7; (Sub2) R2 ← R2 + R6; (Add1) R5 ← -1;	1	1	0	0	1	0	0	0	0	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	26'h6418400
S7	R9 ← R1 x scalingValues[numberOfIteration]; (Mul1) R10 ← R2 x scalingValues[numberOfIteration]; (Mul2)	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	26'h600C0
S8	if[(R3 >3.141592)&(R3 <4.7123)] R9 ← R9 x (-1); (Neg1) R10 ← R10 x (-1); (Neg2)	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	26'h60000
	if[(R3 >1.5707)&(R3<3.141592)] R9 ← R9 x (-1); (Neg1) 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	26'h40000
	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	24'h0

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

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

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

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

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 casued 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.

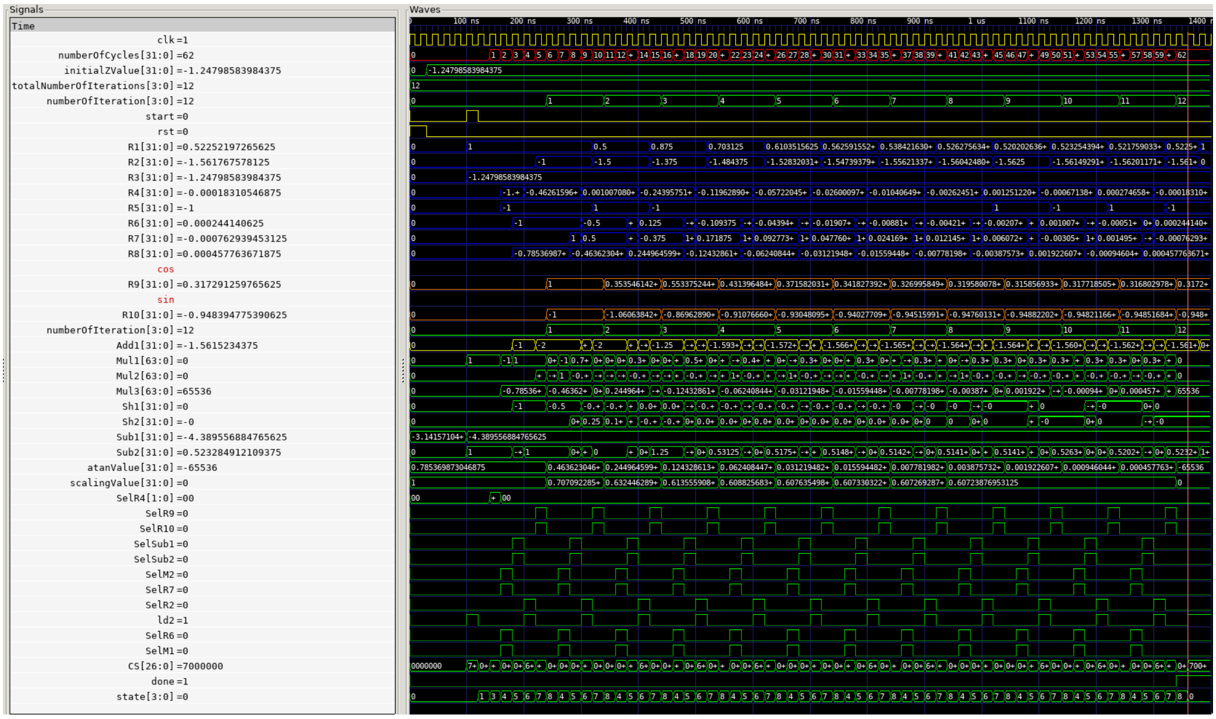

Figure 4 - 4 The whole Verilog simulation of CORDIC algorithm for determining the sinus and cosinus values of angle $\theta = -1.2479$ rad. The value of sinus and cosinus based on the current iteration is also calculated in this algorithm approach. The result is passed to the registers R9 and R10.

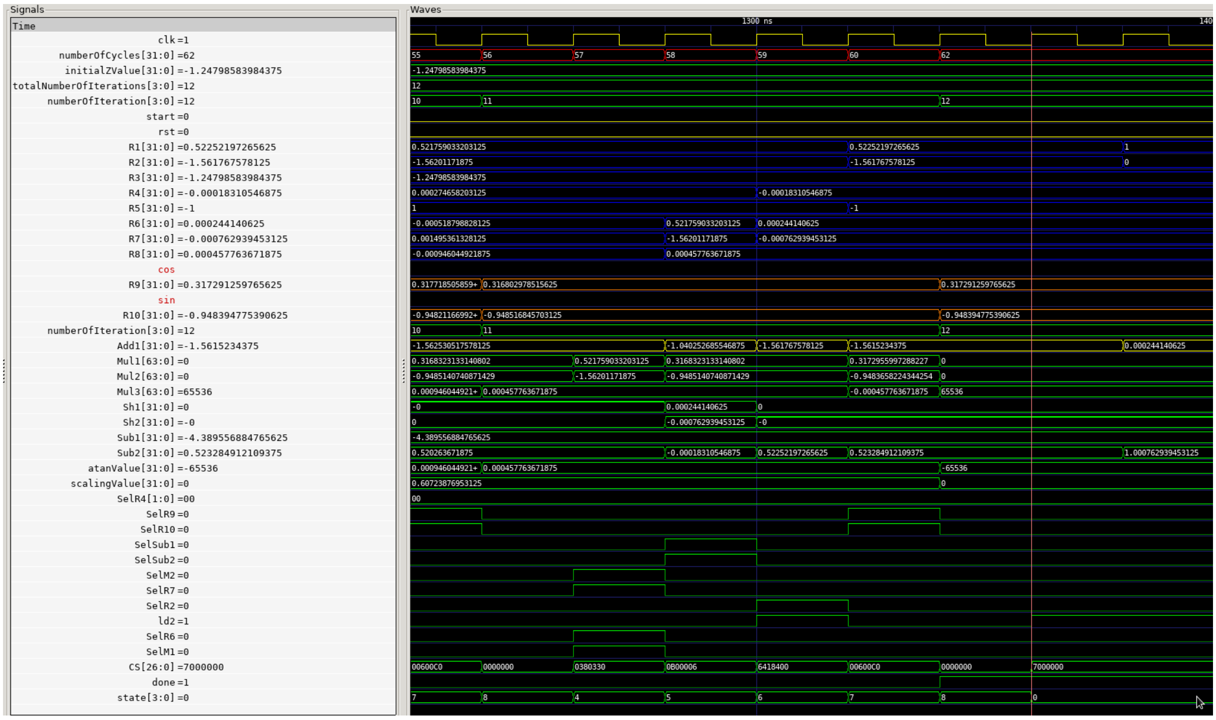

Figure 4 - 5 The detail of the last iteration of the Verilog simulation of CORDIC algorithm for determining the sinus and cosinus values of angle $\theta = -1.2479$ rad. The result is passed to the registers R9 and R10.

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 transcendetal 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 nonlienar equations

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

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

where one possible set of solutions x_1 and x_2 yields

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

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

The algorithm could 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 $\mathrm{F}_1(x_1^0)$ and $\mathrm{F}_2(x_1^0, x_2^0)$

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

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

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

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

As for the general NR algorithm, the inverted value 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.

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

The reciprocal value of the determinant is then calculaetd 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

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

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

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

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

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

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

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

With 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.

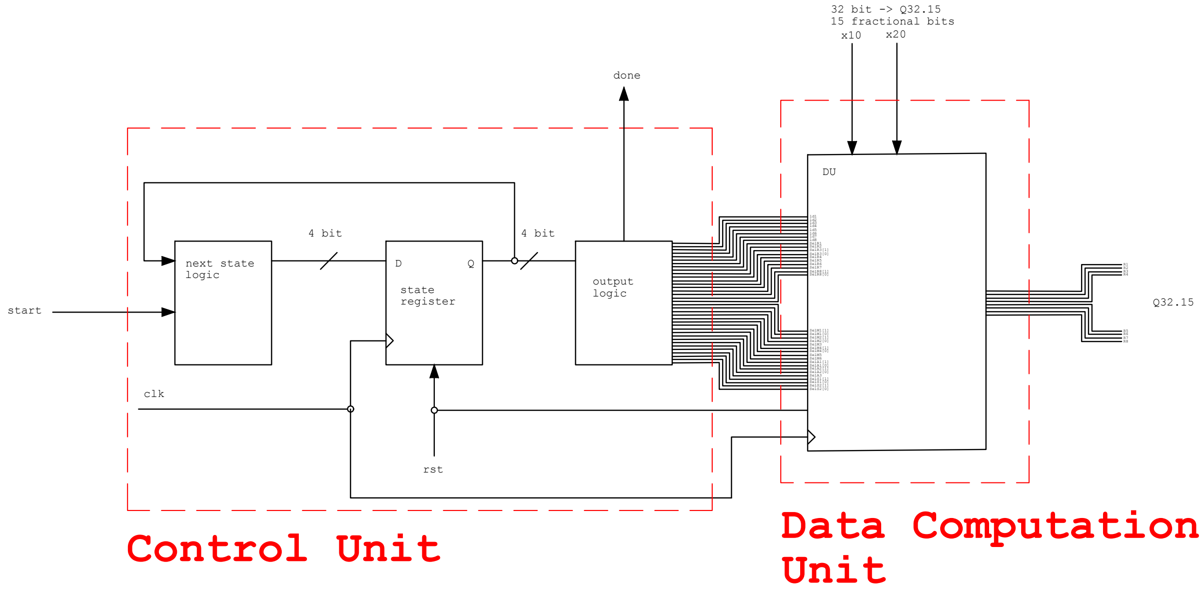

Figure 5 - 1 Top module design for the simple Newton-Raphson (NR) calculation unit module block design.

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.

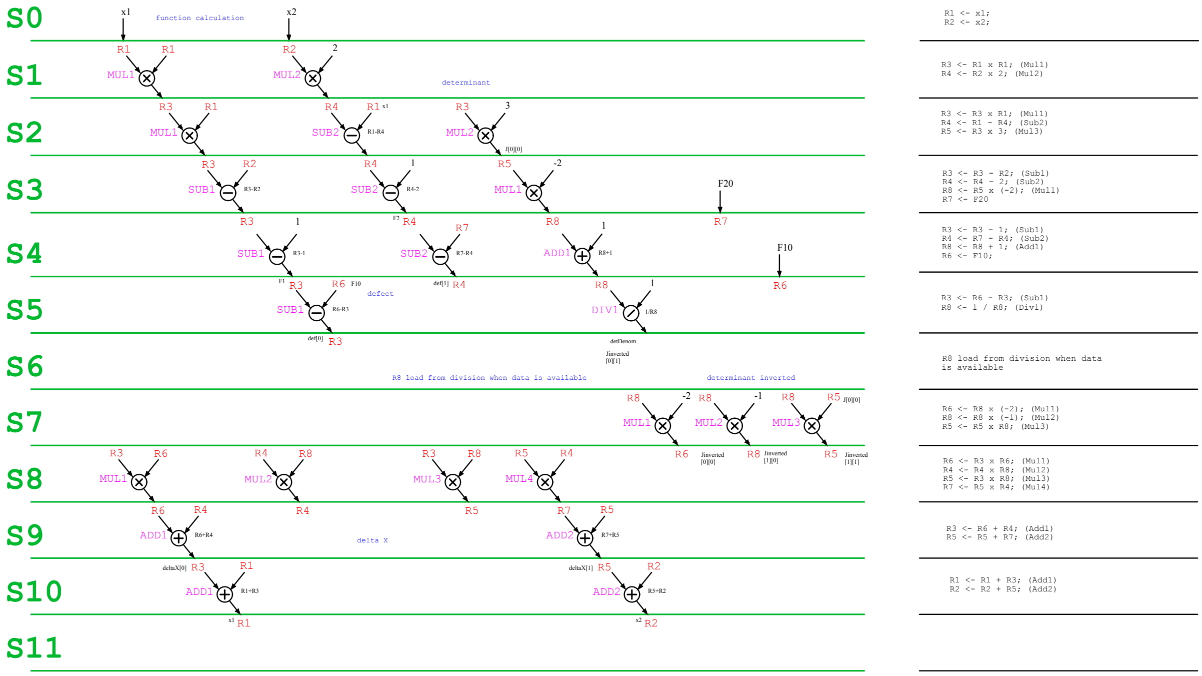

Figure 5 - 2 Allocation and timing diagram for the Data Path Unit part of the simple (NR) module.

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.

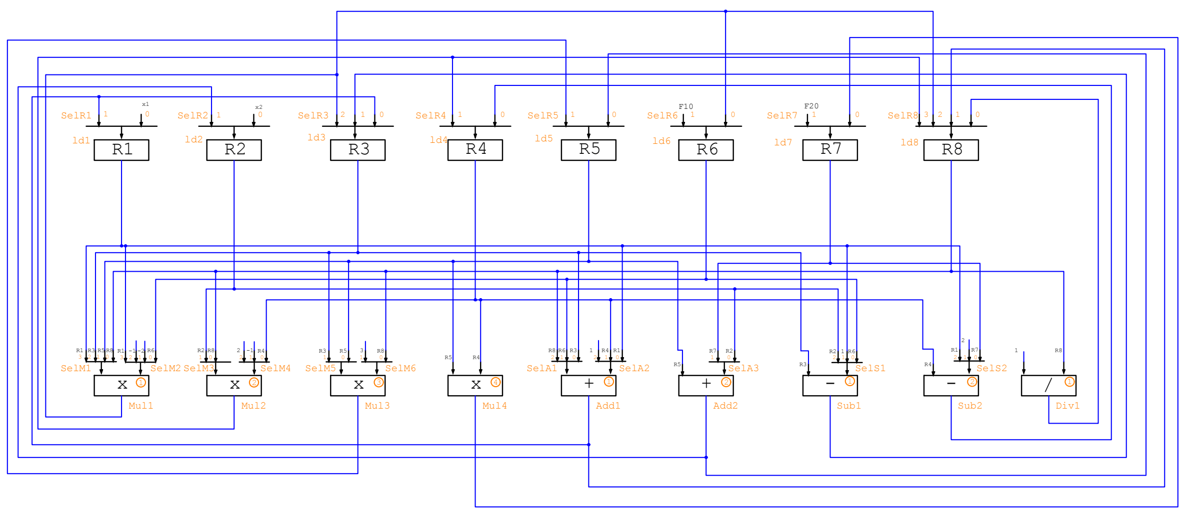

Figure 5 - 3 Register Transfer Level (RTL) scheme of the Data Path Unit part of the simple Newton-Raphson (NR) calculation IP.

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

State	17 SelM1[1]	16 SelM1[0]	15 SelM2[1]	14 SelM2[0]	13 SelM3	12 SelM4[1]	11 SelM4[0]	10 SelM5	9 SelM6	8 SelA1[1]	7 SelA1[0]	6 SelA2[1]	5 SelA2[0]	4 SelA3	3 SelS1[1]	2 SelS1[0]	1 SelS2[1]	0 SelS2[0]	CS
S0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	36’hC00000000
S1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	36’h30283F000h
S2	1	0	1	1	0	0	0	1	1	0	0	0	0	0	0	0	1	0	36’h38242C602
S3	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	36’h331198009
S4	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	1	0	0	36’h351240144
S5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	36’h211000000
S6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	36’h10000000
S7	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	36’hD04C4800
S8	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	36’h1E0C20400
S9	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	0	36’h2800000B0
S10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	36’hC0C000000
S11	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	36’hxxxxxxxxx

5.3 Simulation results

The test bench for simulation was made using Cocotb [1] with the Verilator [2] as a simulator. The 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.

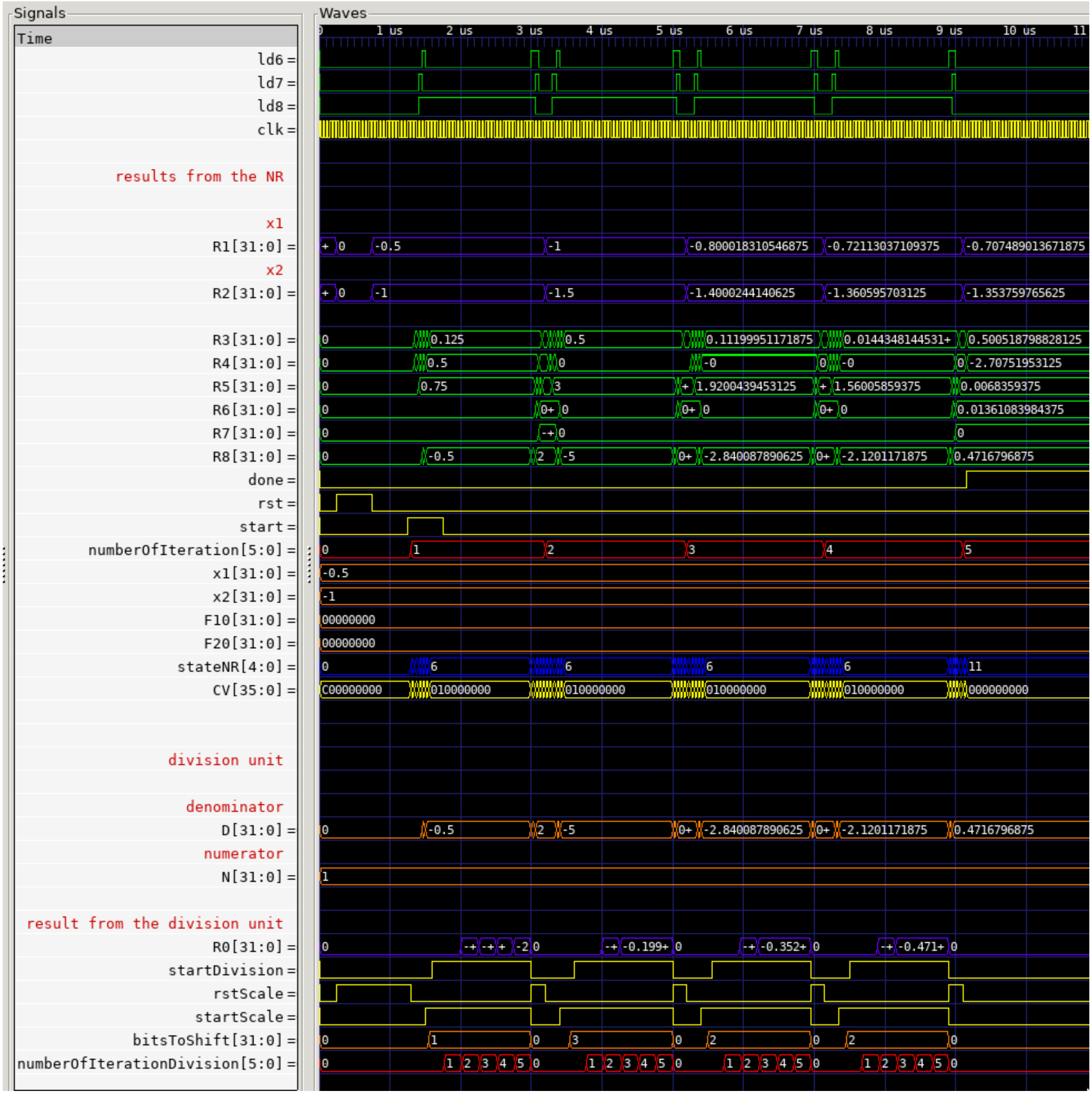

Figure 5 - 4 The whole Verilog simulation of a simple Newton-Raphson (NR) algorithm. The result is may be seen in registers R1 and R2 after the fifth iteration of the algorithm.

6 Selective Harmonic Elimination

6.1 Theory

6.2 High level implementation

The algorithm was for rapid prototyping 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.

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

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

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

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

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

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

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

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

```

```
x1 = x10
x2 = x20

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

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

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


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

    prepDet = prepDet1 - prepDet2

    divDet = 1 / prepDet

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


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

    x1 = x1 + deltaX1
    x2 = x2 + deltaX2

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

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

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

6.3 IP Block Design

6.3.1 Algorithm Block Diagram

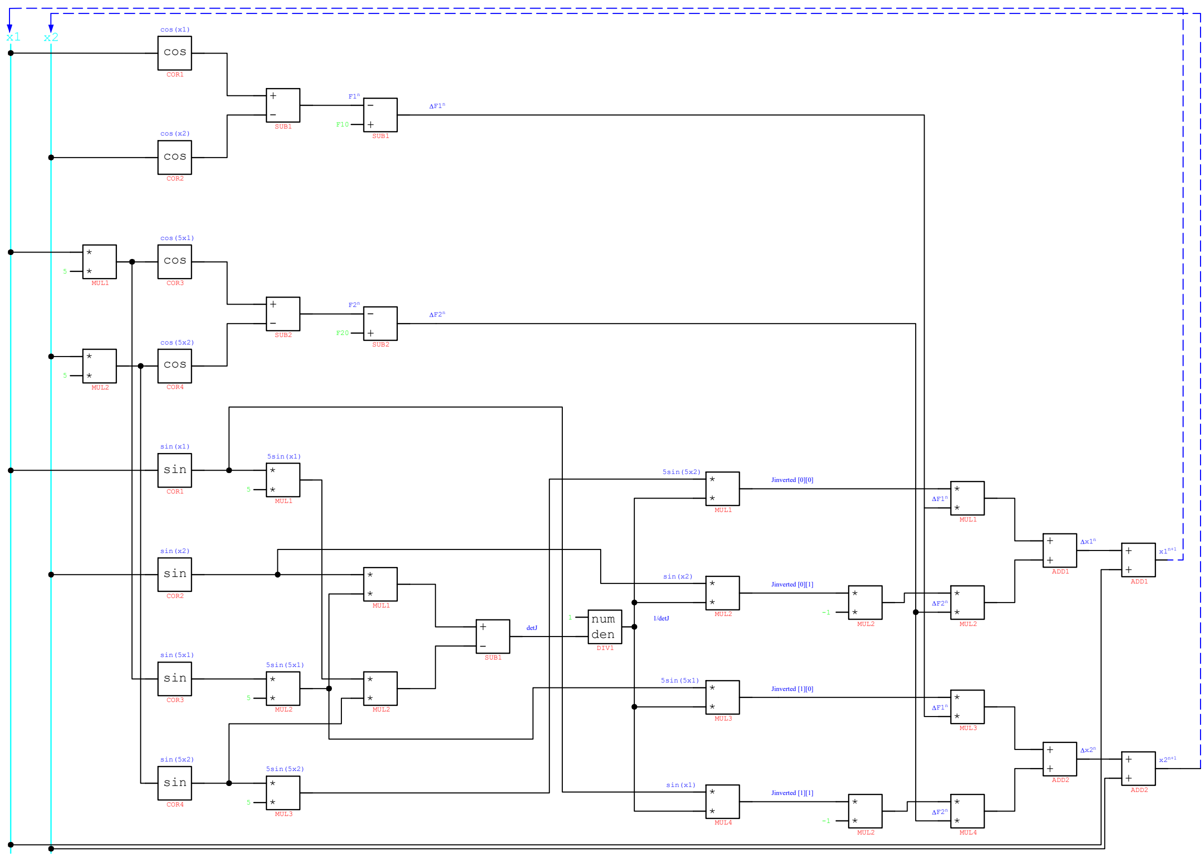

Figure 6 - 1 Block Diagram of the Selective Harmonic Elimination (SHE) using Newton-Raphson algorithm.

6.3.2 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 - 2.

6.3.3 Allocation and Timing

The Allocation and Timing diagram, depicted on Figure 6 - 3 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.

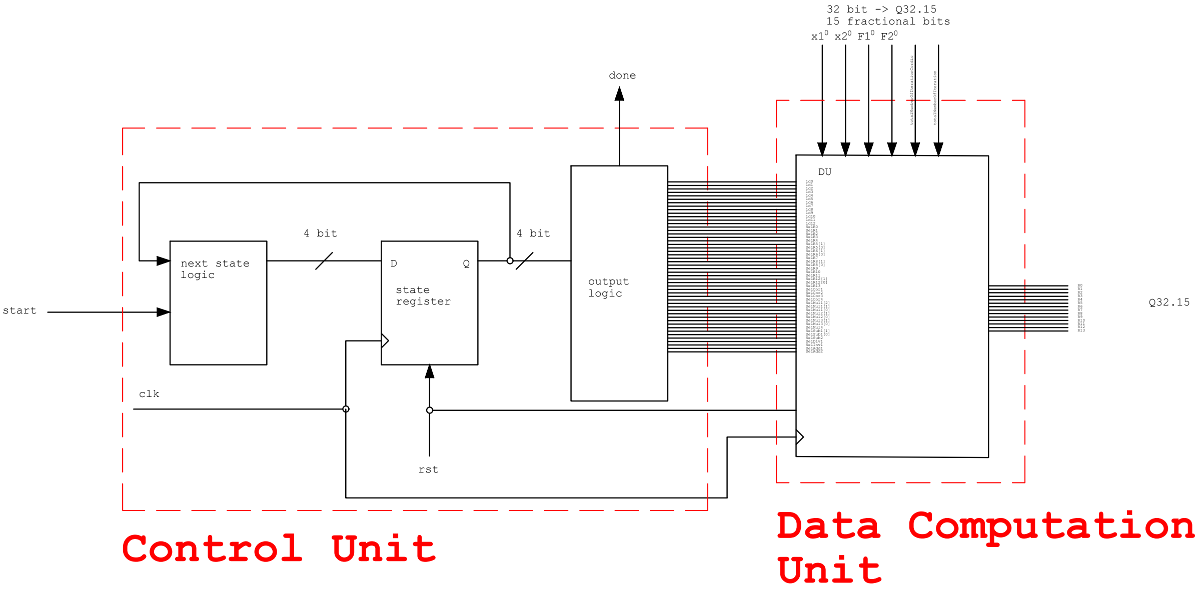

Figure 6 - 2 Top module design for the Selective Harmonic Elimination unit (SHE).

6.3.4 Data Path Unit

6.3.5 Control Unit

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

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

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

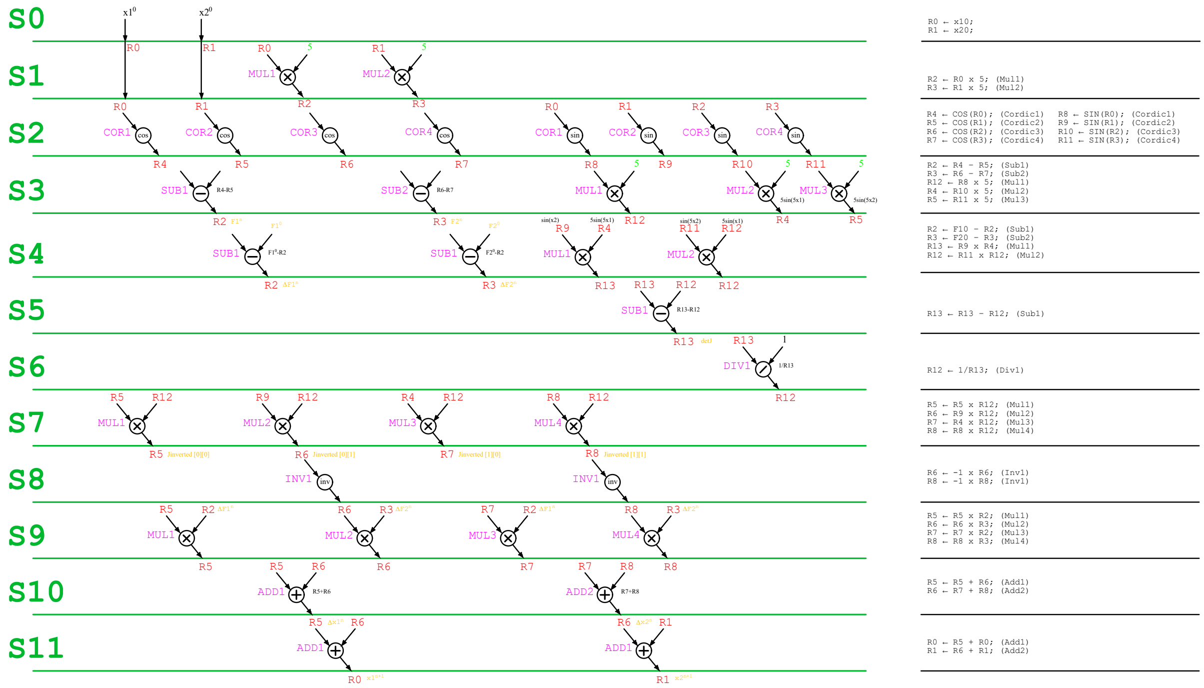

Figure 6 - 3 Allocation and Timing diagram for the Data Path Unit part of Selective Harmonic Elimination (SHE) module.

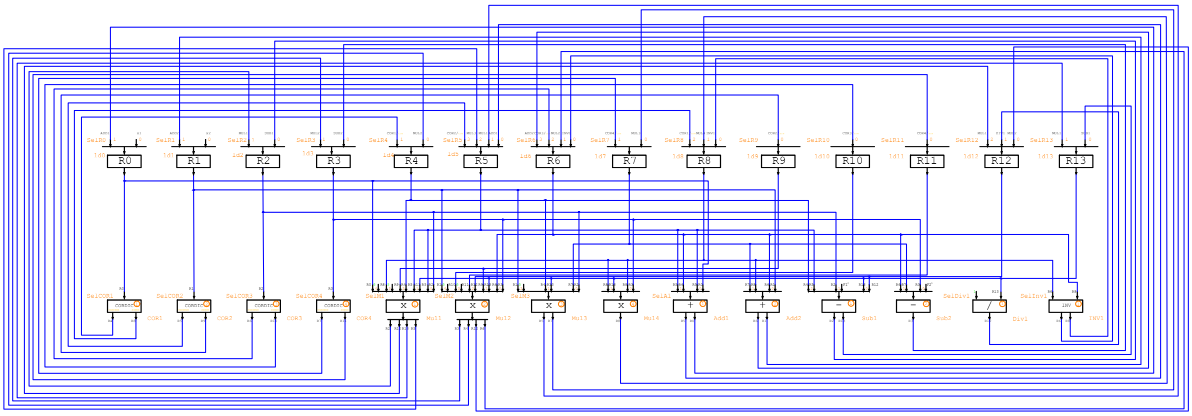

Figure 6 - 4 Register transfer level RTL scheme of the Selective Harmonic Elimination Data Path Unit.

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 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