Czech Technical University in Prague

Faculty of Electrical Engineering

Department of Microelectronics

Design and Evaluation of Algorithms for Fast Power Sensors and Its FPGA Implementation

Master's Thesis

Author:	Bc. Martin Peka
Supervisor:	Prof. Ing. Jiří Jakovenko, Ph.D.
Year:	2024

CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Peka Martin**	Personal ID number:	**492031**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Microelectronics**		
Study program:	**Electronics and Communications**		
Specialisation:	**Electronics**		

II. Master’s thesis details

Master’s thesis title in English:

Design and Evaluation of Algorithms for Fast Power Sensors and Its FPGA Implementation

Master’s thesis title in Czech:

Návrh a zhodnocení algoritmů pro rychlé digitální senzory výkonu a jejich FPGA implementace

Guidelines:

1) Familiarize yourself with algorithms suitable for evaluating mathematical operations on Field Programmable Gate Arrays for power sensors (CORDIC, Newton Raphson interpolation). 2) Create models (in Matlab Simulink) and implement blocks suitable for a power sensor, compare properties and simulation results. The sensor should contain blocks for power computation. In addition, implement blocks for the effective and mean values of measured voltage and current, temperature compensation, overcurrent protection and power factor. 3) Implement the power sensor in HDL (Verilog). If possible, evaluate the designed sensor on a development kit and demonstrate the individual functions.

Bibliography / sources:

[1] Uwe Meyer-Base. Digital signal processing with field programmable gate arrays. 3rd ed. Berlin: Springer, 2007. isbn: 978-3-540-72612-8.
[2] Ray Andraka. “A survey of CORDIC algorithms for FPGA based computers”. In: Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays - FPGA ’98 (1998), pp. 191–200. doi: 10.1145/275107.275139.
[3] DALLY, William J. and HARTING, R. Curtis, Digital Design A Systems Approach. Cambridge University Press, 2012. isbn: 978-0-521-19950-6.

Name and workplace of master’s thesis supervisor:

prof. Ing. Jiří Jakovenko, Ph.D. Department of Microelectronics FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **02.02.2024** Deadline for master's thesis submission: **24.05.2024**

Assignment valid until: **21.09.2025**

prof. Ing. Jiří Jakovenko, Ph.D.	prof. Ing. Pavel Hazdra, CSc.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt Student’s signature

Declaration

I declare that this thesis has been composed solely by myself and all sources used have been cited appropriately.

In Prague, 20. 5. 2024

....

Bc. Martin Peka

Acknowledgements

I would like to thank my supervisor Prof. Ing. Jiří Jakovenko, Ph.D. for valuable advice during the development of this thesis. Also, I would like to express my gratitude to my colleagues from Allegro MicroSystems, Inc., namely to Ing. Miloslav Trnka, Ph.D. who patiently helped me explore the world of digital design engineering, provided guidance and valuable advice throughout my work.

Abstract

This Master's thesis covers the design of a digital power sensor. A brief overview about electrical power affiliated equations is provided. Then several computation algorithms suitable for implementation on FPGA or ASICs are evaluated, and their equations are derived. For a digital power sensor which processes measured signals, especially the reciprocal and square root functions are essential. However other basic functions may be implemented using these computational methods as well. The computational architectures are compared in matter of speed, accuracy, and used hardware resources. For each of these algorithms a simulation is conducted and then the best is chosen for the final implementation into the Zedboard FPGA Evaluation Board. A complete workflow and implementation using the Model Based Design approach from Simulink to bitstream is performed. Final computation on FPGA Evaluation Board of Average power, Apparent power, and effective values from two sampled signals is demonstrated.

Keywords: Newton-Raphson interpolation, CORDIC, Chebyshev approximation, Power sensor, Digital design, Matlab and Simulink, FPGA, Model Based Design

Abstrakt

Tato diplomová práce se zabývá návrhem digitálního sensoru výkonu. Je zde uveden stručný přehled rovnic spojených s elektrickým výkonem. Poté je vyhodnoceno několik výpočetních algoritmů vhodných pro implementaci na FPGA nebo ASIC a jsou odvozeny rovnice, které je popisují. Pro digitální snímač výkonu, který zpracovává naměřené signály, jsou nejpodstatnější zejména funkce dělení a odmocniny. Pomocí těchto výpočetních metod však lze realizovat i další základní funkce. Výpočetní architektury jsou porovnány z hlediska rychlosti, přesnosti a množství použitých hardwarových prostředků. Pro každý z těchto algoritmů je provedena simulace a poté je vybrán nejlepší algoritmus pro konečnou implementaci do FPGA na desce Zedboard. Je ukázán kompletní postup implementace pomocí přístupu Model-Based Design od modelu v Simulinku až po vytvoření bistreamu. Na vyhodnocovací desce s FPGA je demonstrován výpočet průměrného výkonu, zdánlivého výkonu a efektivních hodnot ze dvou navzorkovaných signálů.

Klíčová slova: Metoda tečen, CORDIC, Chebyshevova aproximace, Sensor výkonu, Digitální návrh, Matlab a Simulink, FPGA, Model-Based Design

Contents

List of Figures

Abbreviations

Abbreviations:

CORDIC	Coordinate Rotation Digital Computer
FPGA	Field-Programmable Gate Array
ASIC	Application-Specific Integrated Circuit
DC	Direct Current
AC	Alternating Current
PF	Power Factor
RMS	Root Mean Square
HDL	Hardware Description Language
IC	Integrated Circuit
LSB	Least Significant Bit
MSB	Most Significant Bit
LUT	Look-Up Table
FF	Flip-Flop
DSP	Digital Signal Processor
ADC	Analog to Digital Converter
FIL	FPGA-in-the-Loop
PL	Programmable Logic
PS	Processing System
SoC	System-on-a-Chip
XADC	Xilinx Analog-to-Digital Converter
DRP	Dynamic Reconfiguration Port
JTAG	Joint Test Action Group
AXI	Advanced eXtensible Interface
IP	Intellectual Property
RTL	Register Transfer Level
STA	Static Timing Analysis

1 Introduction

Motivation

Accurate measurement of electrical power and related quantities is important for a large number of applications, from the automotive industry to energetics industry, up to consumer electronics. Computational cores for digital power measurement processing need to address calculation of a few elementary functions, such as the reciprocal, or square root. Since there is increased demand on precision and speed, but every micrometer of silicon is expensive, in this thesis, algorithms used in digital design of ASICs or FPGAs are evaluated to reduce cost, improve speed and accuracy of such algorithms. These algorithms, when correctly designed, provide computational power with minimal hardware resources and high-speed application accurate results. Model Based Design and the connection with FPGA and Matlab in the loop speeds up the development time and minimizes errors with the lower level implementation. This thesis was also developed under supervision from the company Allegro MicroSystems, which is an industry leader in current sensors. Therefore, algorithms used in the digital design as well as the FPGA in the Loop simulation were studied for further design improvements.

Assignment objectives

- Research on the topic of fast computational algorithms for specific functions needed for the signal processing core of a power sensor.
- Modeling of each method and comparison of accuracy, speed, and used hardware resources.
- Implementation of designed core and validation on a FPGA using FPGA in the Loop and Simulink.
- Implemented functions demonstration - measurement and testing with real signals.

2 Power theory

For measuring and monitoring power, several cases are distinguished. There can be DC Power or AC periodic, nonperiodic signals and they also might have some harmonic distortions.

Instantaneous power [1] is voltage multiplied with current as described in equation:1

$$p(t) = v(t) \cdot i(t) \tag{1}$$

This power is equal to the Average Direct Current (DC). For Alternating Current (AC) signals, even if there is no harmonic distortion present and also if the phase shift between the voltage and current is equal to zero, meaning the connected load is purely active, the output power is harmonical and is not constant over one period [2].

Average Electrical Power or Active Power is defined with equation 2

$$P_{avg} = \frac{1}{T} \int_T v(t) \cdot i(t)\ dt \tag{2}$$

Where T is the time of signal observation for the measured signals, v is the measured voltage and i is the measured current together being Instantaneous power which is averaged. The time of signal observation T should be n times the length of the period, otherwise the measurement and computation will introduce methodical error. In discrete form it is a sum of samples divided by N - the number of samples as described in equation 3, [1].

$$P_{avg} = \frac{1}{N} \sum_{n=0}^{N-1} v_n \cdot i_n \tag{3}$$

This equation gives accurate average results for Direct Cunrrent (DC) signals and for Alternating Current (AC) signals with several conditions – Harmonical signals, Active load and current in phase with voltage [1].

In a single phase system, when phase difference between voltage and current occurs, Apparent power and Reactive power are introduced [1]. Here the Active Power can be defined with equation 4

$$P = V_{RMS} \cdot I_{RMS} \cdot \cos(\phi) \tag{4}$$

Where ϕ is the phase difference between the two signals. Apparent power is the product of effective values of current and voltage, given with equation 5

$$S = V_{RMS} \cdot I_{RMS} \tag{5}$$

Where V_{RMS} and I_{RMS} are the root mean squared values defined by equation 6 respectively.

$$x_{rms} = \sqrt{\frac{1}{N} \sum_{i=1}^{N} x_i^2} \tag{6}$$

N is the number of accumulated samples of the corresponding window, x_i is the input signal i.e. voltage or current and x_{RMS} is the calculated effective value.

Reactive power is defined with equation 7

$$Q = V_{RMS} \cdot I_{RMS} \cdot \sin(\phi) = \sqrt{S^2 - P^2} \tag{7}$$

Reactive power is the imaginary part of the measured power.

These powers create a power triangle which can for interpretation be seen on figure 1, diagram taken from [3].

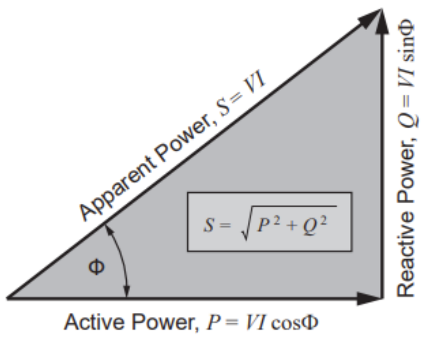

Apparent Power, $S = VI$
Reactive Power, $Q = VI \sin\Phi$
$S = \sqrt{P^2 + Q^2}$
Φ
Active Power, $P = VI \cos\Phi$

Figure 1: Power triangle

Here from the triangle, $\cos(\phi)$ is the Power factor which is defined with equation 8

$$PF = \frac{P}{S} \tag{8}$$

In these conditions, Power Factor PF is equal to the phase shift $\cos(\phi)$. It is the ratio between amplitudes of Active and Reactive power [4].

In Single phase nonsinusoidal situation there is also Distortion and new Deformation power is present. Now the Apparent power consists of three parts. Deformation power or distortion power according to [4] is defined with equation 9

$$P_d = \sqrt{S^2 - P^2 - Q^2} \tag{9}$$

These powers now create a power cuboid instead of a triangle.

Signals in a real situation one phase system consist of a fundamental frequency and different harmonic components which add up to the distortion [1, 2, 3, 4]. Because of this and to have the most universal device, it is reasonable to measure Active Average power which is equal to active power flowing into the device, thus containing the power also with distortions or phase difference. Also, RMS – effective values of measured components give reasonable result. Also, Apparent power might be computed, since it is only a multiplication of RMS values which are already obtained.

3 Model Based Design

This thesis was designed, implemented and executed using the Model based design approach with tools from Mathworks and Xilinx. Model based design allows engineers to increase the development speed and rapid prototyping using higher abstraction level for simulation of prototyped models [5]. These models are defined for example with equations which describe a complex system [6]. Several levels of complexity can describe the model depending on the purpose of each simulation objective. For an overall system functionality simulation in the beginning of development process, the model might consist of less complex equations or approximations in contrast to a model which simulates a transitional behavior in the later development stage. This way it progresses until the detailed final version model is produced [7].

For IC - Integrated Circuit development one approach is to use Matlab and Simulink with HDL Coder [8]. The designer develops a model of a digital circuit in Simulink, simulates and validates the design and then Matlab generates the lower abstraction level HDL code which can be further implemented in a synthesis tool for a FPGA. Other tools also exist, such as from the company Synopsys [7]. The advantage of such an approach is reduced time to debug the design and reduces further verification. The model is systematically re - used through the whole development process. It can also reduce expensive hardware iterations of ASIC devices and reduce time to market for new products. [7, 9]

Workflow and models used in this thesis usually contain a stimulation block, the simulated model and have been compared with an ideal model. The stimuli are first generated from Matlab for debug and development purposes, later real signals are measured but the model persists almost the same and has only slight adjustments for the final implementation.

4 Fixed vs Floating point arithmetic's

There are many numerical representations used in computer science, well know is the floating-point representation providing good dynamic range and easy usage in high level programming languages, or a 2's complement representation of integer which is the most basic interpretation of a number.

Typical integer is not suited for use in calculations where the decimal point is needed and especially when it needs large dynamic range [10].

Floating point representation

The floating-point representation as stated above yields a high dynamic range, meaning it can represent large values as well as very small values. It is composed of two components the significant part of the number and the exponent [11]. The value is represented as equation 10:

$$v = m \cdot 2^{e-x} \tag{10}$$

Where m is the mantissa, e is the exponent and x is the offset of the number which adjusts the dynamic range.

The figure 2 overtaken from [12] explains visually the interpretation:

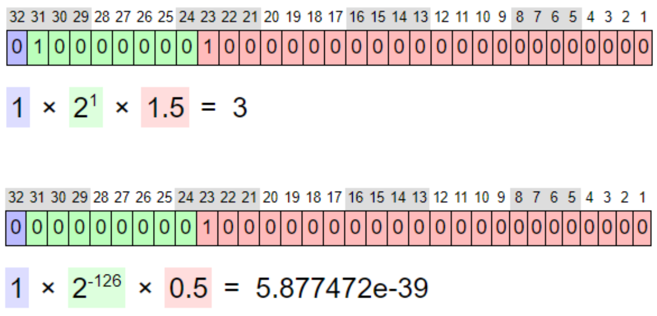


Figure 2: Float example

The MSB bit is the sign, exponent starts from 31st to the 24th bit with the additional feature of the 31st bit which works as a 1 or negative offset of -126 for 32-bit float. The last part is the mantissa, going from MSB as 2^{-1} until the LSB 2^{-23}.

In chip implementation it is common to use a different numerical representation than in high level programming languages such as fixed point arithmetic. Many desired algorithms on FPGA work properly in small intervals typically in range of $[0.5, 1)$ or $[0.25, 2)$.

The main difference between fixed point and floating point representation is its tradeoff between speed and dynamic range and precision. Double is also easier to use for complex algorithms [10], [13].

Fixed point

In this number representation the decimal separator has its fixed position. The first bit of the number is the sign and then there are several bits before the decimal separator and several bits after. Fixed point can be expressed as following range:

$$\frac{[-2^{n-1};\ 2^{n-1}-1]}{2p} \tag{11}$$

Where n is the *n-bit* number with p bits after decimal separator.

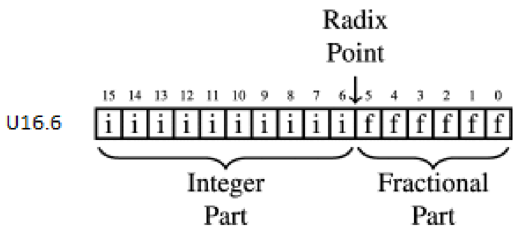


Figure 3: Fixed point example

Which best illustrates the figure 3 above overtaken and modified from [14].

The fixed point data type in Matlab and Simulink has the following notation - fixdt(s, w, f) in short notation $Sw.f$, where S is the singedness, w is the total word length and f is the numeric of fractional bits behind the decimal separator.

A big advantage of this representation is that processors work with this numeric format the same way they work with normal integers, so the amount of operations is reduced, they are simplified and much more faster. The disadvantage is in the lost range. If a larger number needs to be interpreted, it is at the expense of lost precision in the fractional part and vice versa [15]. When adding two fixed point numbers, they must be represented in the same format and the result is represented by the same format + 1 bit in the word length. A similar situation occurs when multiplying two fixed point numbers, where the new format is the combination of the two multiplied numbers i. e. for the integral part it is the sum of the left bits before the decimal separator and similarly for the fractional part. With fixed point arithmetic's overflow of number may occur, especially if the number of bits in front of the decimal separator is reduced for additional resolution of the fractional part. Therefore, it appears advantageous to use special formats such as $S12.10$ or $S12.11$ where in the second format the first bit is used as the sign bit, it has integral bits and 11 fractional bits creating interval between $[-1, 1-\mu_{LSB})$. The resolution is equal to 2^{-LSB}, in this case $\frac{1}{2048}$. When multiplying using these number formats, the overflow cannot occur, because the result of product of numbers at the interval $[-1, 1)$ stays at the interval $(-1, 1]$.

5 Elementary function computation methods

For all algorithms presented in this section, theoretical background with derivation of necessary equations is shown. For the simulation and implementation in Matlab Simulink of the described technique and comparison to the ideal value, the function Square Root is chosen. At the end of this section, a comparison of studied algorithms in terms of precision, speed, and used hardware resources is conducted.

CORDIC

Derivation of equations for the trigonometric functions

CORDIC stands for Coordinate Rotation Digital Computer, it is a hardware efficient and fast iterative algorithm for calculating a wide variety of functions. It's principle is in the usage of transformation between Polar and Cartesian coordinates and only using shifts and adders ([16]).

The algorithm was first invented for the use in the B-58 aircraft for precise high accuracy navigation system as a replacement for the old analog [17].

After this research it was further evolved into computation of other functions. In several modes one can compute hyperbolic functions as well as square root, division, and reciprocal.

To obtain the sin or cos value of the desired angle ϕ. In each step, addition or subtraction of a predetermined angle value is performed depending on the sign where if the new rotated angle is larger or smaller than the given angle. These predetermined angle values have known x and y values which are the *cosine* or *sine* values. Using this iteration process if the angle values are half the previous angle value it converges to the solution. It is similar to the Bisection method. A number can be interpreted as a point in Cartesian coordinates. It can be expressed as its magnitude and angle. So, for the derivation of the CORDIC algorithm for calculation of trigonometric functions, the initial equations are the rotation transformation from cartesian to polar coordinates [16], [18].:

$$x_0 = |A| \cdot cos(\phi) \tag{12}$$
$$y_0 = |A| \cdot sin(\phi) \tag{13}$$

When adding an angle, it changes into

$$x_1 = |A| \cdot cos(\phi + \delta) \tag{14}$$
$$y_1 = |A| \cdot sin(\phi + \delta) \tag{15}$$

Now using the angle sum identity

$$sin(\phi + \delta) = sin(\phi)cos(\delta) + cos(\phi)sin(\delta) \tag{16}$$
$$cos(\phi + \delta) = cos(\phi)cos(\delta) - sin(\phi)sin(\delta) \tag{17}$$

We get for x

$$x_1 = |A| \cdot cos(\phi + \delta)$$
$$x_1 = |A| \cdot [cos(\phi)cos(\delta) - sin(\phi)sin(\delta)]$$
$$x_1 = |A| \cdot cos(\phi)cos(\delta) - |A| \cdot sin(\phi)sin(\delta) \tag{18}$$

Plugging in $x_0 = |A| \cdot cos(\phi)$ we get

$$x_1 = x_0 \cdot cos(\delta) - y_0 \cdot sin(\delta) \tag{19}$$

And similarly for y

$$y_1 = y_0 \cdot cos(\delta) + x_0 \cdot sin(\delta) \tag{20}$$

Now for the first simplification, using the trigonometric identities

$$cos(\delta) = \frac{1}{\sqrt{1 + tan^2(\delta)}} \tag{21}$$
$$sin(\delta) = \frac{tan(\delta)}{\sqrt{1 + tan^2(\delta)}} \tag{22}$$

the equations 19 and 20 can be rewritten to

$$x_1 = x_0 \cdot \frac{1}{\sqrt{1 + tan^2(\delta)}} - y_0 \cdot \frac{tan(\delta)}{\sqrt{1 + tan^2(\delta)}}$$
$$y_1 = y_0 \cdot \frac{1}{\sqrt{1 + tan^2(\delta)}} + x_0 \cdot \frac{tan(\delta)}{\sqrt{1 + tan^2(\delta)}}$$

For x_1:

$$x_1 = \frac{1}{\sqrt{1 + tan^2(\delta)}} \cdot (x_0 - y_0 \cdot tan(\delta))$$
$$x_1 = \frac{1}{\sqrt{1 + tan^2(\delta)}} \cdot (x_0 - y_0 \cdot tan(\delta))$$

$$x_1 = cos(\delta) \cdot (x_0 - y_0 \cdot tan(\delta)) \tag{23}$$

And y_1:

$$y_1 = cos(\delta) \cdot (x_0 + y_0 \cdot tan(\delta)) \tag{24}$$

If the rotation is limited only to one (first) quadrant and also if it limits the rotations so that $tan(\delta) = \pm 2^{-i}$, the tangent part can be implemented only using shifting. This is for $i = 0..N = 45°, 26.5°, 7.125°$. cos($\phi$) is identical within the first quadrant, it can be considered as a constant expression which will $cos(-\phi)$ times scale the value in the last step. If a fixed amount of iterations is used for given required precision, the value of the constant expression can be precomputed and hardwired in the chip or FPGA thus reducing the area and adding more simplification. We get:

$$x_{i+1} = K_i[x_i - y_i \cdot d_i \cdot 2^{-i}] \tag{25}$$
$$y_{i+1} = K_i[y_i + x_i \cdot d_i \cdot 2^{-i}] \tag{26}$$

Where

$$K_i = \cos(\arctan(2^{-i})) = \frac{1}{\sqrt{1+2^{-2i}}} \tag{27}$$

$$d_i = \pm 1 \tag{28}$$

d_i is the direction which decides if the angle is added or subtracted. The algorithm scales the result by A_n each step:

$$\Pi_i \sqrt{1+2^{-2i}} \tag{29}$$

The sum of rotated composite angles can be stored in a similar accumulator equation, which is also defined by arctangent. Result from this equation is the total rotation after i iterations.

$$z_{i+1} = z_i - d_i \cdot \arctan(2^{-i}) \tag{30}$$

In conclusion from the derivation above the full set of equations is as follows:

$$x_{i+1} = K_i[x_i - y_i \cdot d_i \cdot 2^{-i}] \tag{31}$$

$$y_{i+1} = K_i[y_i + x_i \cdot d_i \cdot 2^{-i}] \tag{32}$$

$$z_{i+1} = z_i - d_i \cdot \arctan(2^{-i}) \tag{33}$$

Where

$$d_i = -1 \text{ if } z_i < 0 \text{ or } d_i = +1 \text{ if } z_i > 0 \tag{34}$$

[17] describes Cordic in two modes, the first is called Rotation which rotates the input vector by a specified angle.

The second mode of operation is called *Vectoring*, where the input vector is rotated in such direction so that the vector is aligned with the x axis, y is equal to zero. The sign is calculated from the y value which the algorithm tries to minimize. The result is then the angle value rotated by the algorithm. In the third differential equation for the angle accumulator if the initial value is zero, then at the end it contains the angle. For the Vectoring mode, only equation 34 adjusts while everything else stays the same [19].

To fulfill the initial condition of the first quadrant, correction rotation of $\pm\frac{\pi}{2}$ needs to be performed.

Extension into hyperbolic function

For this thesis, extension into hyperbolic functions is important, because the computation of square root is needed in the proposed sensor architecture, and it has a complex (split complex) relation to hyperbolic functions. Because of the similarities between hyperbolic and trigonometric functions, the hyperbolic functions can be derived with the similar identities and simplifications resulting into similar equation differentiating in one sign:

$$x_{i+1} = K_i[x_i + y_i \cdot d_i \cdot 2^{-i}] \tag{35}$$
$$y_{i+1} = K_i[y_i + x_i \cdot d_i \cdot 2^{-i}] \tag{36}$$
$$z_{i+1} = z_i - d_i \cdot \arctan(2^{-i}) \tag{37}$$

Where

$$d_i = -1 \text{ if } z_i < 0 \text{ or } d_i = +1 \text{ if } z_i > 0 \tag{38}$$

However according to [20] Hyperbolic functions need certain iterations repeated to converge to the appropriate result [16]. For $K \to \inf$: Iterations $= (4, 13, 40 \ldots 3K + 1)$.

Proposed CORDIC Core for Square Root

For the purpose of square root computation, a fixed point Model has been created in Simulink, which simulates a digital core with CORDIC algorithm derived above. It can be seen in figure 4.

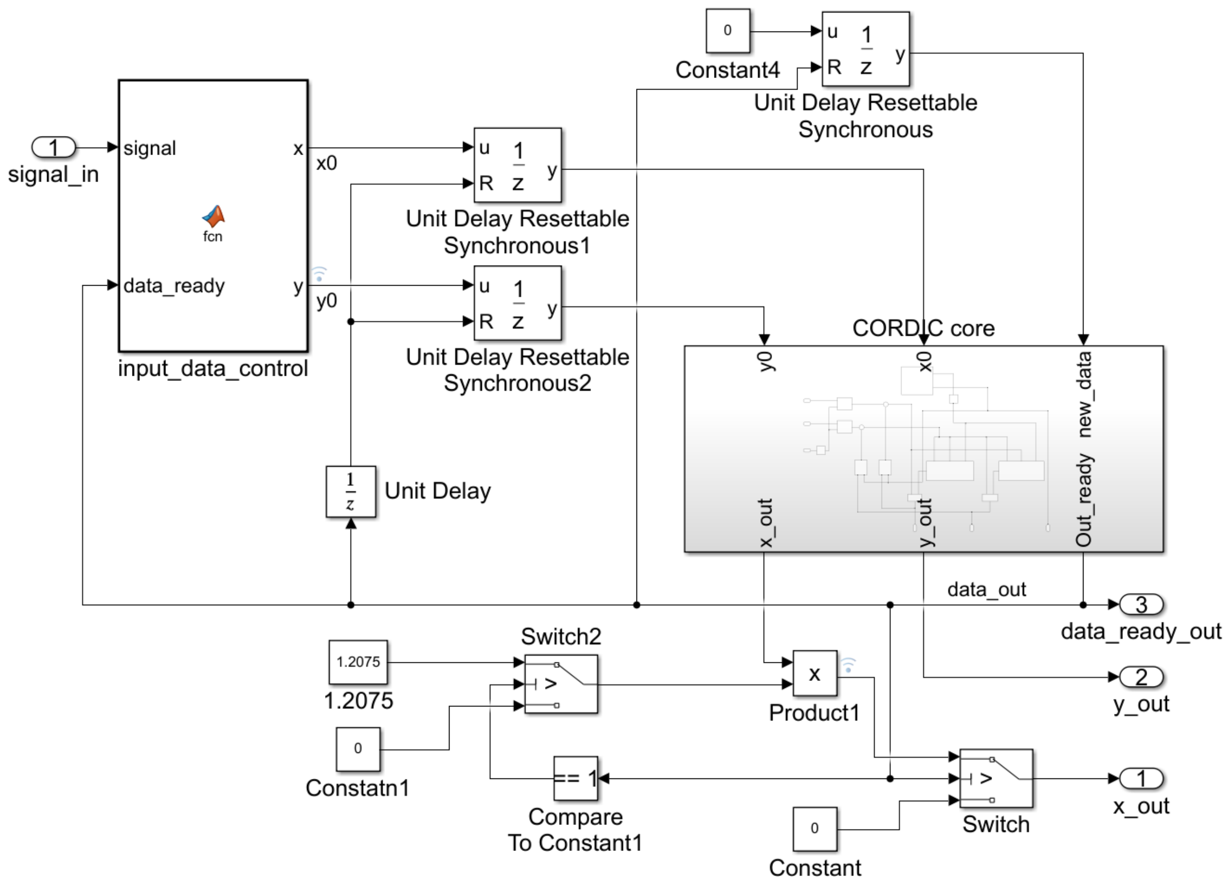


Figure 4: CORDIC Core Top level

To find the square root of a number s, the initial condition must be properly set. The input number must be scaled to the interval $[0.5, 1)$. Then the signal is decomposed into two signals, $x = s + 0.25$ and $y = s - 0.25$ which in this model is represented as *signal_in*. The Matlab function block *input_data_control* controls the input data and decomposes the signal. These signals are stored in the *Unit Delay Resettable Synchronous 1* and *2*, which serve as input registers for the *CORDIC core*. This block is responsible for shifting and adding operations and is in figure 5. After the computitation in the *CORDIC core*, the result value is scaled with the precomputed constant 1.2075 if the boolean output *Out_ready* is true. Also this resets the the input registers and the control subsystem, which is prepared for new input signals.

The main subsystem which computes the square root is in figure 5.

A finite state machine *counter_shifting_logic_fixed* controls the process of iterations and proper number shifting. In the first state, x and y values are introduced into the differential equation constructed using the sum block and the feedback path. On each iteration the shift block shifts the input data and assigns the proper sign. In the add block the operation is completed, and the signal is saved in the feedback unit delay block. If the number of iterations exceeds 14, flag bit *Out_ready* is set high, which allows the top level to scale the output and allows new data into the CORDIC evaluation subsystem. The finite state machine also performs the additional iterations. This occurs on the 4th and 13th iteration, where it needs to perform the shift and add operation again but without increasing the number of bits to shift, to ensure the convergence to the result. In total with 16 cycle, it achieves 14 bit precision.

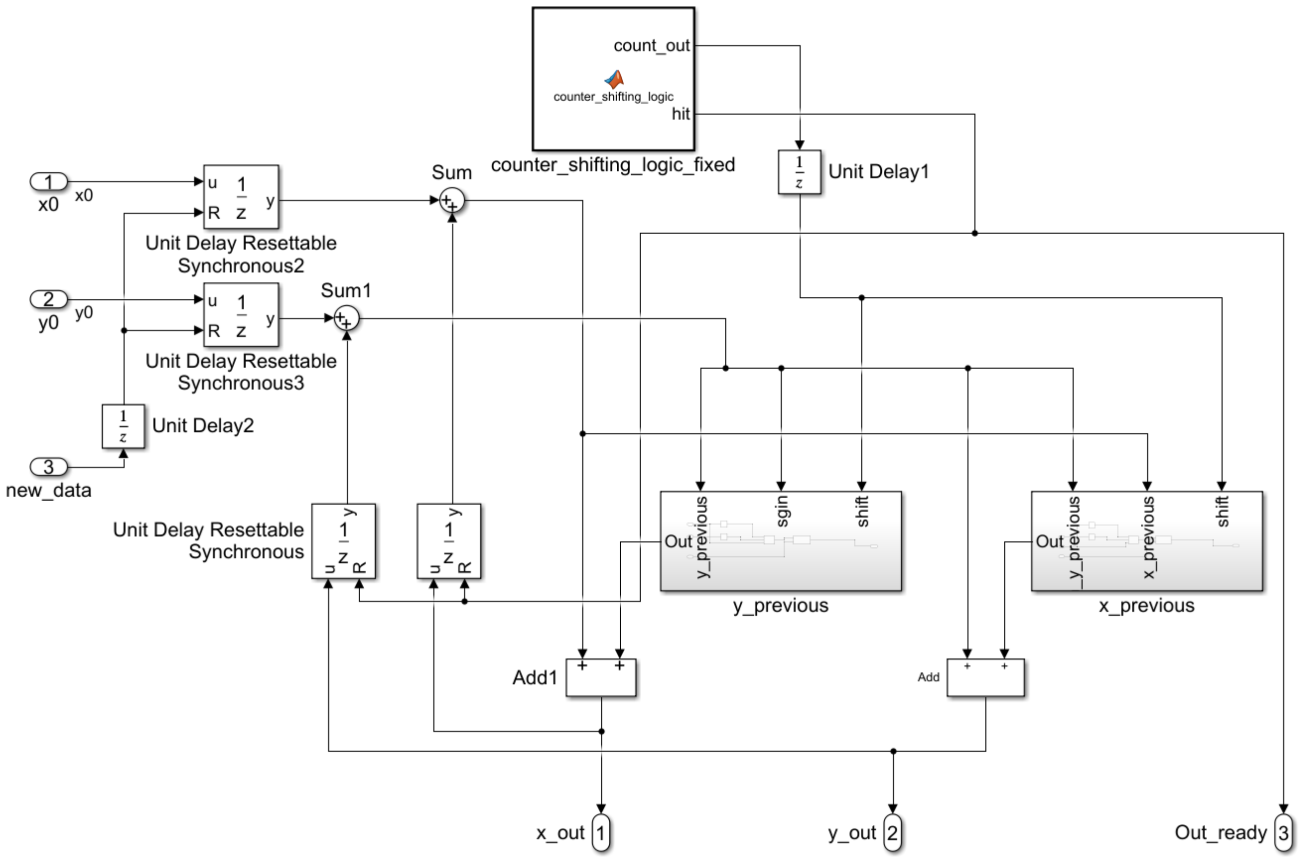


Figure 5: CORDIC Core

In figure 6 can be see that the fixed point simulation has an error between $-3 \cdot 10^{-5}$ and $3 \cdot 10^{-5}$ which is a span of $6 \cdot 10^{-5}$. The *precision* rounded to 15 bit is calculated from equation 39.

$$precision = \log_2 |3 \cdot 10^{-5}| = -15.0247 \tag{39}$$

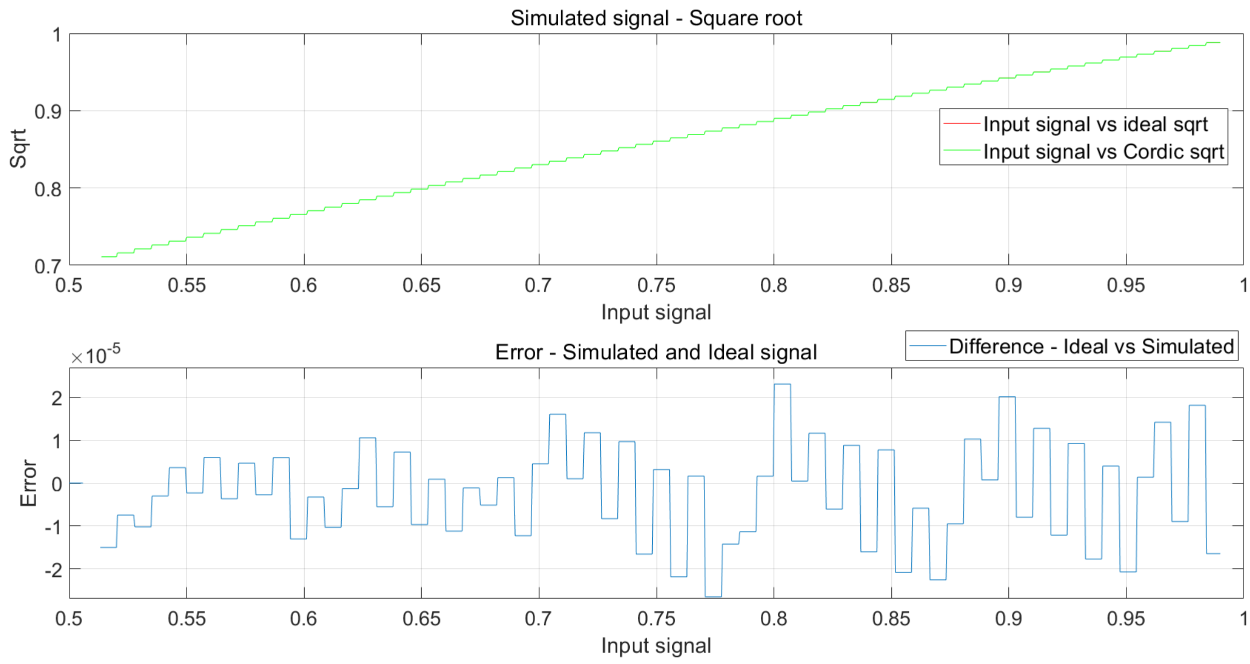


Figure 6: CORDIC Simulation

Newton Raphson Interpolation

Newton Raphson theory and equations

The Newton Raphson interpolation method is a fast root finding iterative algorithm that can quickly converge to a solution in a small number of steps with high accuracy. This means, that with each iteration the bit precision is doubled. In this thesis it is used to calculate the square root or to calculate the reciprocal, which can then be used to calculate division.

The general equation for the discrete Newton Raphson interpolation is

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{40}$$

Square root is usually calculated from the inverse square root, because it eliminates divison, which would require a lot of computational power for the division operator[21].

For the inverse square root, the initial value a will be

$$x_i = \frac{1}{\sqrt{a}} \Longleftrightarrow \sqrt{a} = \frac{1}{x_i} \Longleftrightarrow a = \frac{1}{x_i^2} \tag{41}$$

Then the function is

$$f(x_i) = \frac{1}{x_i^2} - a \tag{42}$$

And the derivative

$$f'(x_i) = -\frac{2}{x_i^3} \tag{43}$$

When equations 42 and 43 are plugged into the initial equation 40 we obtain

$$x_{i+1} = x_i - \frac{\frac{1}{x_i^2} - a}{-\frac{2}{x_i^3}} \tag{44}$$

$$x_{i+1} = x_i + \frac{x_i^3 \cdot (\frac{1}{x_i^2} - a)}{2} \tag{45}$$

Which looks simplified as

$$x_{i+1} = x_i + \frac{x_i}{2}(1 - a \cdot x_i^2) \tag{46}$$

This equation 46 is suitable for hardware implementation as it uses only a multiplier and bit shift [22].

After few iterations when solution with required precision is found, the final value x_i is multiplied by the initial searched value because the inverse square root value $\frac{1}{\sqrt{x_i}}$ was found but the searched value was the square root $\sqrt{x_i}$

$$x \cdot \frac{1}{\sqrt{x}} = \frac{\sqrt{x^2}}{\sqrt{x}} = \sqrt{x} \tag{47}$$

Initial estimation highly increases the convergence speed thus reducing the number of iterations. Several approaches may be used for initial estimation for example look up table or function approximation. The closer the initial approximation is, the higher precision is obtained and less iterations are required [21].

The method is suitable for fixed point implementation. The input interval is restricted to $[0.5, 2)$ or to $[0.25, 1)$ and prescaling needs to be performed. These intervals are convenient, because prescaling can be done only by dividing 2^k and than rescaling back using $2^{\frac{k}{2}}$. However, interval in range $[0.5, 1)$ is better for the Chebyshev precomputation for the Initial estimation, which can be seen in figure 7. Precision of the precomputation on the $[0.25, 1)$ interval is 5 bits, and on $[0.5, 1)$ 8 bits. Theoreticaly after two iterations of the NR equation, the precision for the first interval is 20 bits, but for the second 32 bits.

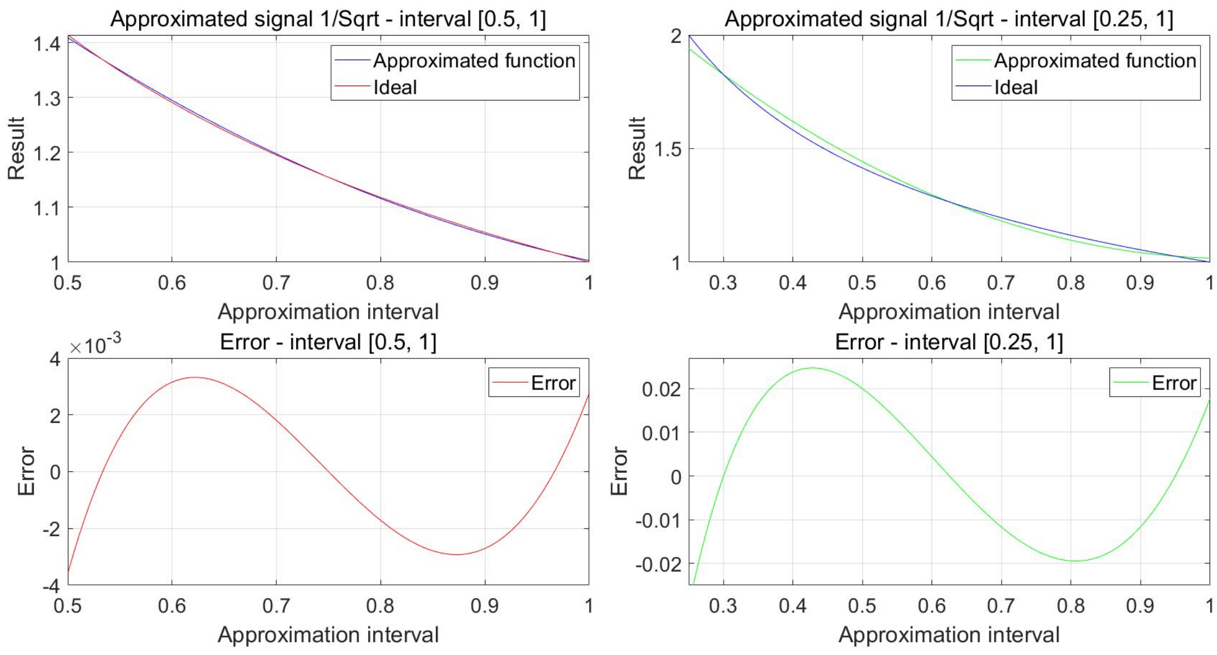


Figure 7: Chebyshev precomputation for Newton Raphson Inverse Square Root Comparision between interval $[0.5, 1)$ and $[0.25, 1)$.

However, if the fixed number interval in range $[0.5, 1)$ is used, additional steps in prescaling need to be performed. It is usually done by dividing 2^k and again, the output number must also be rescaled to return appropriate result. The correction is then performed by a factor of $\sqrt{2^k}$. For odd numbers this is equal to $2^{\frac{k}{2}}$ and for even $\sqrt{2} \cdot 2^{\frac{k-1}{2}}$, where k is the number of bits shifted in the initial prescaling section.

Similarly for reciprocal numbers the Newton Raphson method is used. Here the function is

$$f(x_i) = a \cdot x_i - 1 \tag{48}$$

And the derivative

$$f'(x_i) = \frac{1}{a} \tag{49}$$

Then plugging these two equations 48 and 49 into the initial equation 40 it results in

$$x_{i+1} = x_i - \frac{1}{a}(a \cdot x_i - 1) \tag{50}$$

The term $\frac{1}{a}$ is equal in the current iteration to x_i, so the equation is simplified to

$$x_{i+1} = x_i - x_i(a \cdot x_i - 1) \tag{51}$$

Which can be further simplified to the final equation 52 for computation of the reciprocal value:

$$x_{i+1} = x_i \cdot (2 - a \cdot x_i) \tag{52}$$

The reciprocal value also needs to be prescaled, but the resulting interval is not rescaled, so the final value needs to be rescaled by the same amount of bits as it was prescaled [23].

Proposed Newton Raphson inverse square root model

The proposed structure in figure 8 uses 2 iterations and a precomputation block in figure 9. The precomputation is performed using function approximation with found chebyshev coefficients 5. The initial estimation is the inverse function of square root with a minimal precision of 7 bits. For example, for a *input value* = 0.5 the Newton Raphson method finds the solution with a precision around 30 bits for ideal model and 28 bit when limited by the fixed point implementation which in this case is 4 bits for word part and 28 bits for fractional part, so in total 32bits. The first iteration has a precision of 16 bits. The second then has 28 bit precision.

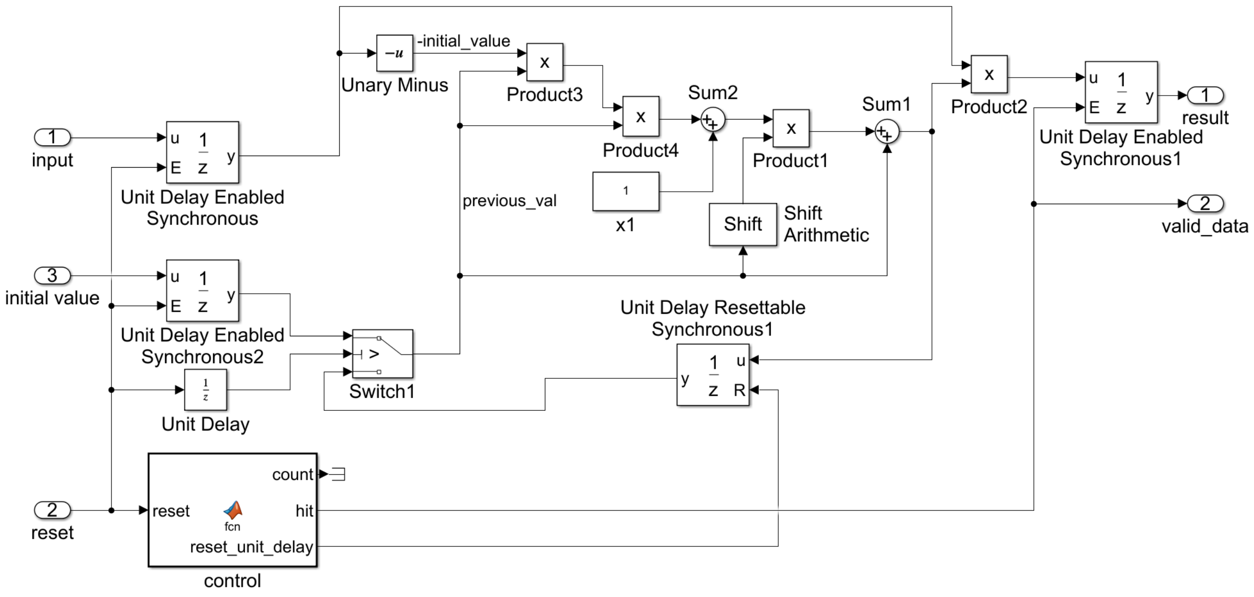


Figure 8: Newton Raphson inverse square root

In figure 8 the state machine *control* block controls the whole iteration process. The *reset* input resets the whole submodule and both *input* and *initial value* are propagated to the delay blocks. Then the equation 46 is performed and the solution is stored in the *Unit Delay Resettable Synchronous1* block. Then the process is repeated with the previously computed value. After the second iteration, the value is multiplied by the input value and result is produced.

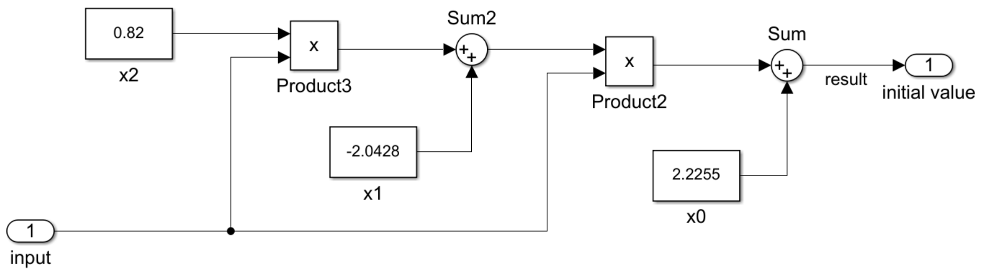


Figure 9: Newton Raphson pre estimation

The preestimating submodule approximates the inverse square root value with two multiplication and three addition blocks reordered according to the Horner's method 5. The output value is then passed as the initial estimation to the Newton Raphson block. The output value x is described using equation 53.

$$x = 2.2255 + x_i \cdot ((0.82 \cdot x_i) - 2.0428) \tag{53}$$

Because of the preestimation uses values outside the interval $[-1, 1)$, additional bits in the word length must be added.

In figure 10 is the fixed point Newton Raphson simulation with error between $-5.5 \cdot 10^{-9}$ and $4 \cdot 10^{-9}$. The *precision* is around 27 - 28 bits and is calculated from the same equation 39 as in CORDIC. Precision vareis also because of the fixed point implementation.

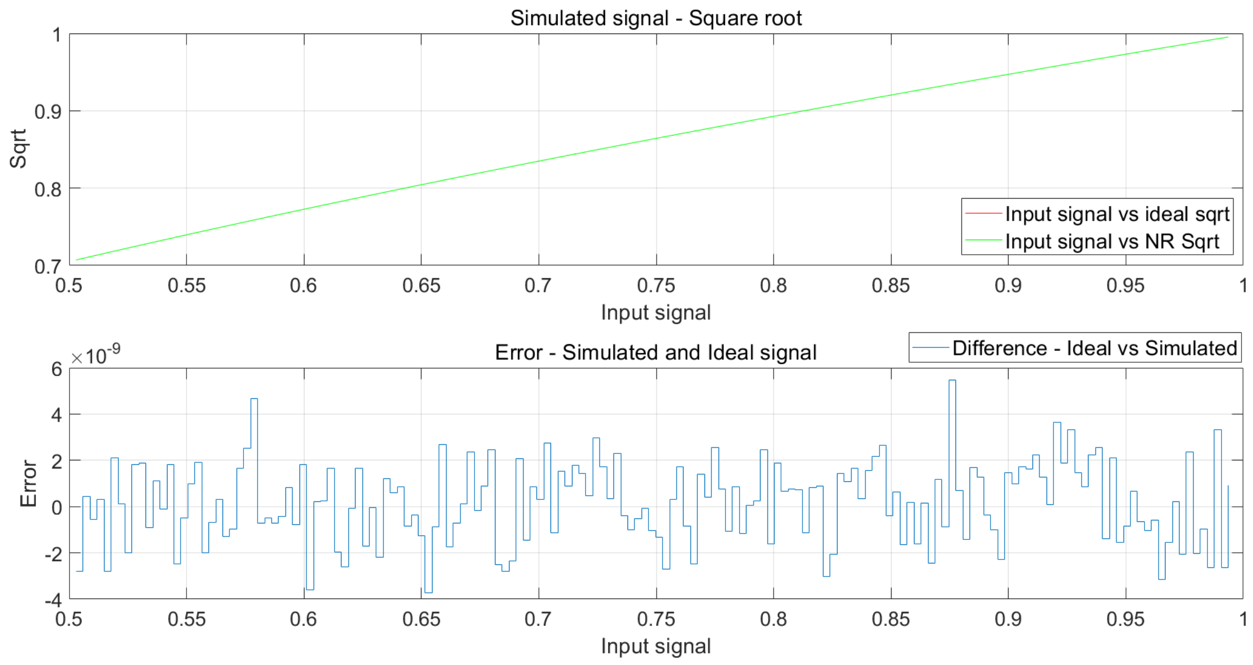


Figure 10: Newton Raphson interpolation result

Chebyshev aproximation of functions

Approximation theory

Using the Chebyshev polynomials, it is possible to approximate functions [24]. With this process the constants are precomputed, and the function approximation is much faster than using iterative algorithms stated above. The area implementation should be larger, since multiplication is used instead of adding and shifting as for the CORDIC algorithm.

Chebyshev polynomials of the first kind are defined as:

$$T_k(x) = cos(k \cdot \arccos(x)) \tag{54}$$

On the interval

$$-1 \leq x \leq 1$$

Using the trigonometric identities, it can be shown that we can interpret Chebyshev polynomials as normal not trigonometric[24].

The calculation is possible to obtain using the Moivre's theorem or it can be computed directly from definition.

$$\begin{gathered} T_0(x) = cos(0 \cdot \arccos(x)) = 1 \\ \text{and} \\ T_1(x) = cos(1 \cdot \arccos(x)) = x \end{gathered}$$

For higher order polynomials, a recursive formula has been found, given by the equation:

$$T_{n+1} = 2xT_n(x) - T_{n-1}(x), \text{for } n \geq 1 \tag{55}$$

The first 6 polynomials of the first kind computed from the recursive equation:

$$\begin{aligned} &T_0(x) = 1, \\ &T_1(x) = x, \\ &T_2(x) = 2x^2 - 1, \\ &T_3(x) = 4x^3 - 3x, \\ &T_4(x) = 8x^4 - 8x^2 + 1, \\ &T_5(x) = 16x^5 - 20x^3 + 5x, \\ &T_6(x) = 32x^6 - 48x^4 + 18x^2 - 1 \end{aligned}$$

Using these polynomials, it is possible to approximate a large variety of functions including square root, reciprocal, trigonometric functions and so on. Chebyshev polynomials have the advantage against the Taylor series, that they need less coefficients and thus polynomial order for the same approximation accuracy [24]. Because of the definition interval $[-1, 1]$ a transformation of the approximation interval needs to be performed using the equation:

$$\bar{x} = \frac{1}{2}[(b - a)x + (a + b)] \tag{56}$$

Functions are approximated in the Chebyshev nodes giving the minimax criterium [25] from this equation:

$$x_i = \cos\left(\frac{2i-1}{2n}\pi\right), i = 1, \ldots n \tag{57}$$

Searching for the minimax criterium is the task of finding the best dc approximation of a function in a given interval. Minimizing the maximum absolute value of the difference from the desired function on the given interval gives the nodal points from the recommended cosine distribution.

After that the functional value at the nodal points of the approximated function are obtained. These then represent the solution vector. The polynomials are inserted into the Vandermonde matrix [26] in such a way that they are calculated at the function point corresponding to the approximated value. The equation is solved by multiplying by an inverse matrix, and the solution is the individual weights of the partial polynomials.

The resulting polynomial is then defined by the equation:

$$\sum_{j=1}^{N-1} c_j T_j(x) \tag{58}$$

The square root or other functions such as reciprocal used in the effective value definition may be approximated using polynomials. According to [27] Chebyshev polynomial approximation has few advantages in contrast to Taylor polynomial approximation. Mainly the approximated function may be computed using much fewer coefficient thus achieving higher precision with fewer coefficient which means fewer multiplication blocks in the design. According to [27] only 4 coefficient are required with the Chebyshev approximation for 16 bit precision, however using the Taylor series with 4 coefficients only 4 bits of precision is achieved.

In the listing 1, a script which computes insverse square root function approximation with the third order polynomial can be seen. Also symbolic reordered into the Horner Method is shown. Other functions may be aproximated as well with this script, only by changing the line 13 in the code to other reference functions.

```
clear;
N = 3;
a = 0.5;
b = 1;

for i = 1:N
    xi(i) = cos( pi()*(2*i-1)/(2*N) );
end

xi = 0.5 * ( a + b ) + xi * 0.5 * ( b - a );
xi = xi'

y = 1./sqrt(xi)

vect1 = [1, xi(1), 2*xi(1)^2-1];
vect2 = [1, xi(2), 2*xi(2)^2-1];
vect3 = [1, xi(3), 2*xi(3)^2-1];

D = [vect1; vect2; vect3]

coef = D \ y

%% Symbolic
syms x
format long
polynom = sum([coef(1)*1, coef(2)*x, coef(3)*(2*x^2-1)])
c = sym2poly(polynom)

%% Horners Method
polynom_horner = horner(polynom)
c = sym2poly(polynom_horner);
fprintf("C2: %f , C1: %f  , C0: %f\n", c(1), c(2), c(3));
```

Listing 1: Example script of Chebyshev Approximation of $\frac{1}{\sqrt{x}}$

Proposed Chebyshev structure for Square root

The Chebyshev approximation of the square root is implemented is in figure 11. The input value is evaluated with the approximated coefficients of the Chebyshev polynomial series. The series has been rewritten with the Horner's Method 5 to a more compact and signal reusing form.

Using the calculations stated above written into a Matlab script for finding the optimal Chebyshev coefficient, resulted in finding the same coefficients as from [27].

In comparison with the CORDIC or Inverse square root using Newton Raphson method, this approximation is more compact and elegant only using two multipliers and two sums, which both may already be implemented in the FPGA DSP slices.

The found coefficients on interval $[0.5, 1)$ are $C_0 = 0.2183, C_1 = 0.8801$ and $C_2 = -0.0988$. The approximated value may then be calculated as:

$$x = 0.2183 + 0.8801 \cdot x_0 - 0.0988 \cdot (2x_0^2 - 1) \tag{59}$$

Rewritten with the Horner's Method:

$$x = 0.3171 + x_0 \cdot ((-0.1976 \cdot x_0) + 0.8801). \tag{60}$$

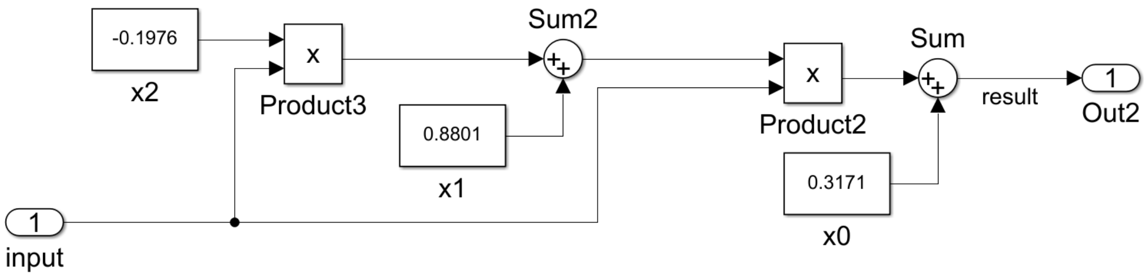


Figure 11: Chebyshev Implementation

In figure 11 The input signal is driven into two multiplication blocks and two additions thus exactly copying the equation 60.

According to the simulation in figure 12 the error is between $6 \cdot 10^{-4}$ and $-6 \cdot 10^{-4}$. The precision calculated from equation 39 from maximum absolute error is 10 bits. In the error simulation 4 extremes can be seen, which corresponds to the minmax criterium, where the principle of finding the best approximation should result into $n + 1$ maximal deviations from the ideal value, where n is the order of the polynomial.

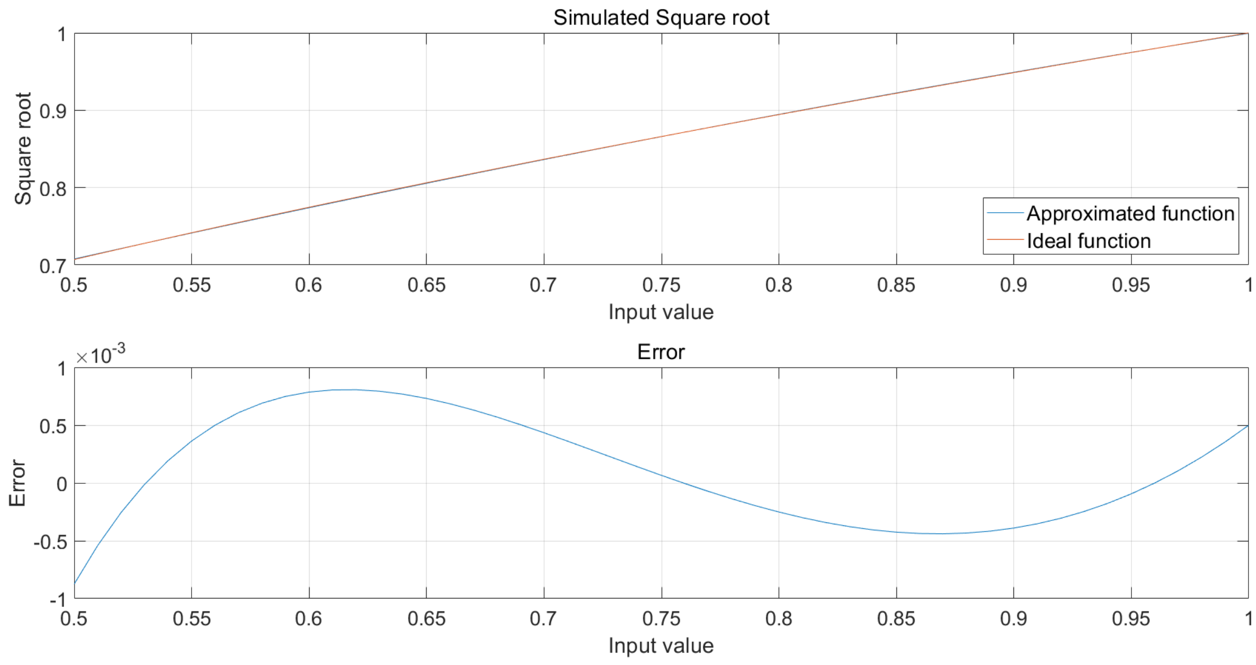


Figure 12: Chebyshev Simulation

Horner's Method

Every polynomial is possible to rewrite using the Horner's Method including the Chebyshev series [28]. The Horner's method was invented as an optimalization for polynomial division.

For every polynomial given by

$$p(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \tag{61}$$

the polynomial can be rewritten into the form as follows [28]:

$$p(x) = a_0 + x\left(a_1 + x\left(a_2 + x\left(a_3 + \cdots + x(a_{n-1} + x a_n)\right)\right)\right) \tag{62}$$

These coefficients can be precomputed and hardwired with the multiplying blocks into the FPGA thus leading to low computational requirements and low component requirements. This method can also be used with iterative formulas such as the Newton Raphson interpolation method [27].

The difference between monomial polynomial evaluation is in the amount of computational blocks which is $\frac{(n^2+n)}{2}$ if powers are calculated individually and n additions or $2n - 1$ and n additions, if the powers are reused. In contrast the Horner's method uses only $\frac{n}{2} + 2$ multiplications and n additions.

Comparison of researched Algorithms

In table 1 comparison between different algorithms of used hardware resources, achieved precision, and speed requirement is presented for the above implemented square root evaluation. Implementation results are obtained from the Vivado Design Suite, further explained in section 8, where brief introduction to the software tool and synthesis, implementation is described.

As expected, the CORDIC algorithm occupies a fewer amount of DSP slices, since it has fewer multiplications, but uses much more logic for its computation. Precision and speed is fixed to the number of cycles within one function evaluation. For the Newton Raphson algorithm, the precision depends on the preestimated input precision, thus $2 \cdot x$ in the table, but generally as mentioned in section *Newton Raphson theory and equations* 5 it doubles the precision with each clock cycle and the solution is rapidly found within few clock cycles. The Chebyshev approximation occupies the same amount of DSP slices as the NR algorithm, but does not have any FFs, since it produces the output within one clock cycle. Implementation issues with setup and hold time between input and output registers might easily occur here, so additional pipelining - Flip Flops between the multipliers might be added. Precision is fixed, depending on precomputed coefficients, and used multipliers.

Comparison of Algortihms		LUT	FF	DSP	Precision (bit)	Clock cycle
CORDIC	Synthesis	641	135	4	14	$14 + 2$
	Implementation	640	167	4		
Newton Raphson	Synthesis	204	131	8	$2 \cdot x$	$2 + 2$
	Implementation	204	163	8		
Chebyshev approximation	Synthesis	120	0	8	8	1
	Implementation	120	0	8		

Table 1: Comparison table of hardware resources, precision, and speed between different algorithms. *LUT* stands for Look Up Tables, *FF* for Flip Flops and *DSP* for digital signal processor.

However, the CORDIC core can be much more easily modified to support multiple functions, whereas the Newton Raphson is fixed. The Chebyshev approximation could also evaluate other functions if the polynomial coefficient would be stored in a look up table and changed with added logic to compute the desired function approximation.

For the purpose of designing a fast power sensor, combination of Newton Raphson interpolation with Chebyshev coefficients precomputation is chosen, because together they achieve satisfactory precision within a few clock cycles.

6 Proposed Power sensor architecture

Design overview

The proposed sensor computes average power from the equation 3. The sensor would receive data from an AD Converter which would sample voltage and sample for example output from differential Hall plates or from a shunt resistor for current [3]. Also, it computes the effective values of these two measured values given by the equation 5. From these two signals it computes the Apparent power so the difference between the real active power and the distortions and or phase shift difference can be observed. To achieve the evaluation of above mentioned equations, the reciprocal and square root functions need to be computed. Reciprocal is used for the term $\frac{1}{N}$ which is the number of accumulated samples. This value is used in both the Average Power and also in both of the effective values.

The signal is sampled over a window, which might be adaptively defined, and the sensor recalculates and adjusts the number of stored samples. The window does not have an overlap, because it would require more accumulation. An approach with recalculating the results on each sample would require a multi - rate clock domain with a sufficiently higher computational core speed in contrast with the sampling circuit. The effective values are calculated over the same period and thus also the apparent power.

For both functions the Newton Raphson interpolation has been chosen but with an initial estimate of the function using the Chebyshev approximation third order polynomial. Using this combination, it is possible to achieve precision of 32 bits in two clock cycles, 8 bits precision with the precomputation and with two iterations of Newton Raphson which each time doubles the precision to 32 bits. Also, several shifters are used and an accumulative sum.

Top level

The proposed top level design can be observed in figure 13. The input signals are in format $S17.16$ and are labeled *signal_in_i* and *signal_in_v* in the Simulink schematic. Other inputs are the *window_length* and a flag value to recompute the window reciprocal value labeled *calculate*.

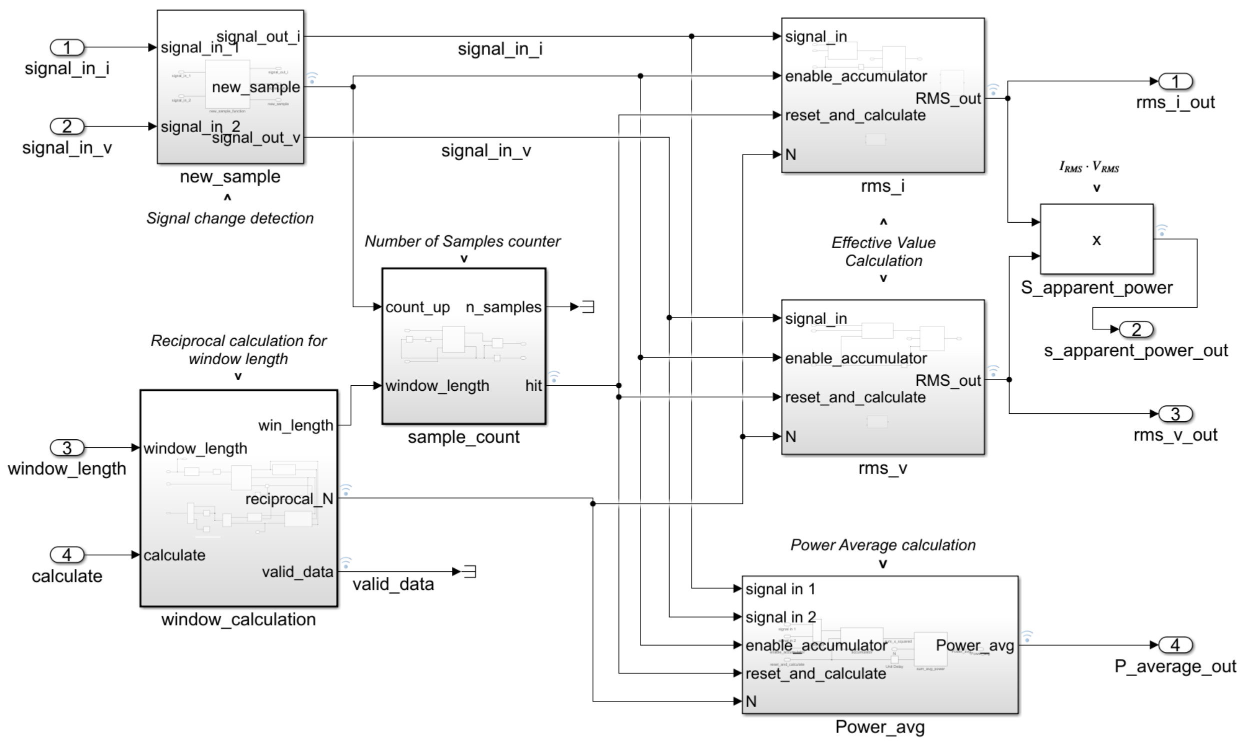


Figure 13: Power sensor - Top level

Outputs are *rms_i_out*, *s_apparent_power_out*, *rms_v_out* and *P_average_out* which correspond to the outputs of each specific computation block. The output format of these four outputs is $S24.20$.

In the top level seven block can be seen. The first block - *new_sample* is responsible for signal change detection. It counts the amount of samples which are stored in each accumulator hidden in the rms and power blocks. The input signal is sampled into the *new_sample* block which holds and puts a flag that a new sample arrived. Then on each *new_sample* flag, *sample_count* block counts upwards the number of samples stored in the accumulators.

Sample count block

The *sample_count* module is a simple counter, which counts on each *count_up* signal until the count number equals the *window_length* value and then toggles the *hit* flag which serves as a trigger for *rms_i*, *rms_v* and *Power_avg* blocks for their calculation. It also resets itself with a one clock cycle delay. The submodule can be seen in figure 14.

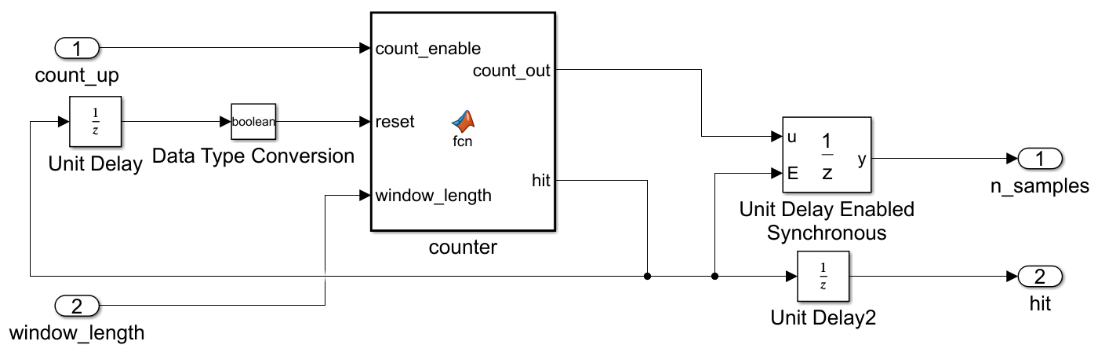


Figure 14: *sample_count* block scheme

Window calculation block

The *window_calculation* block as mentioned above is responsible for finding the reciprocal of the number of samples, which is then used in other block for computing the responsible equations 3 and 6. For this task, the Newton Raphson algorithm has been chosen with Chebyshev polynomial approximation for the initial estimate. First the input value of the window length must be shifted to the appropriate interval. An initial fixed shift of 6 bits is performed, because the default window of minimum 200 samples will be shifted at least 8 times. Then this number is put into the *shifter* block, which shifts the integer until it is in the desired range. The scheme of this block can be observed in the section 6 in the figure 22. The number of shifted bits is stored and later used for the correction of the reciprocal value.

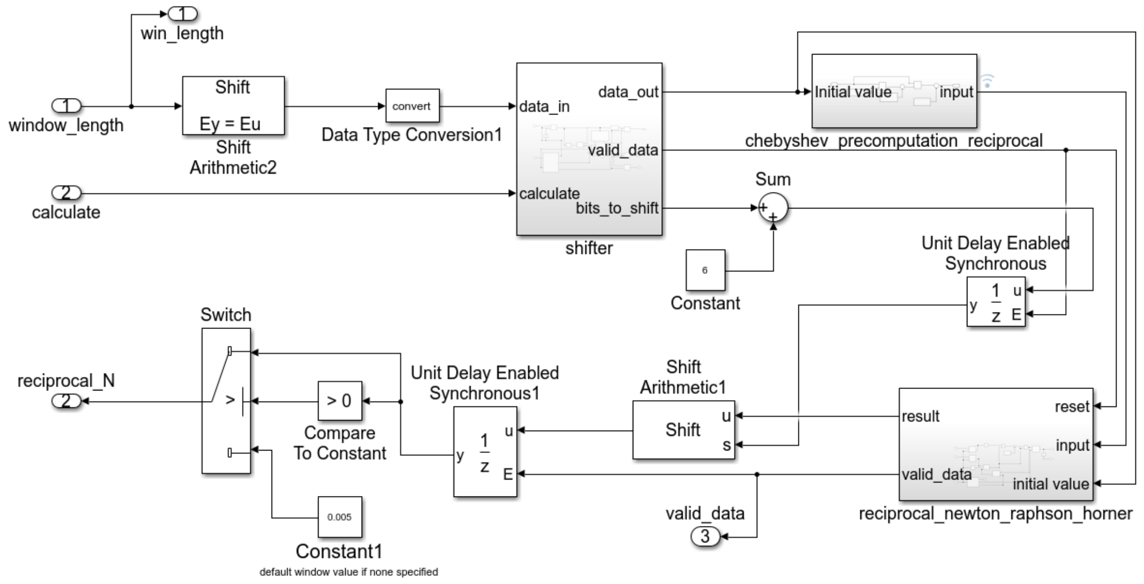


Figure 15: *window_calculation* block

Precomputation for the reciprocal Newton Raphson method

The precomputation submodule is best described using a equation 63 as in section 5. The coefficients have been found with the same script as for the square root approximation, with the change to find the reciprocal value. The script might be seen in the Appendix. The initial estimation is the reciprocal value with a precision of 8 bits.

$$x = 4.242424 + x_0 \cdot ((2.5859 \cdot x_0) - 5.818181). \tag{63}$$

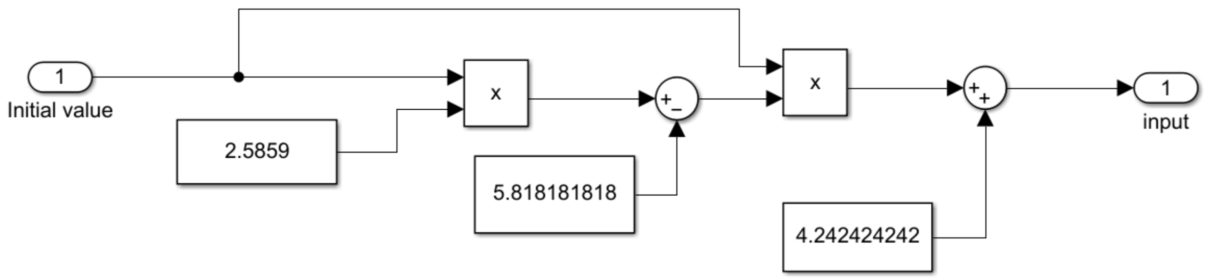


Figure 16: *chebyshev_precomputation_reciprocal* block

Reciprocal Newton Raphson block

The main calculation is done in the submodule *reciprocal_newton_raphson_horner*. In this block the equation 52 is performed. In two iterations, the reciprocal value is found. The *control* block is responsible for controlling the iteration process. First it stores *input* value and the *initial value* into D type flip flops. Then it performs the first round of the equation with the *input* value and stores the result into a third D type flip flop. Then in the second round it performs the same calculation only this time using the previously calculated value instead of the initial input. The *initial value* input remains the same. On the fourth clock cycle, the result is stored in the output unit delay and a valid data signal is asserted.

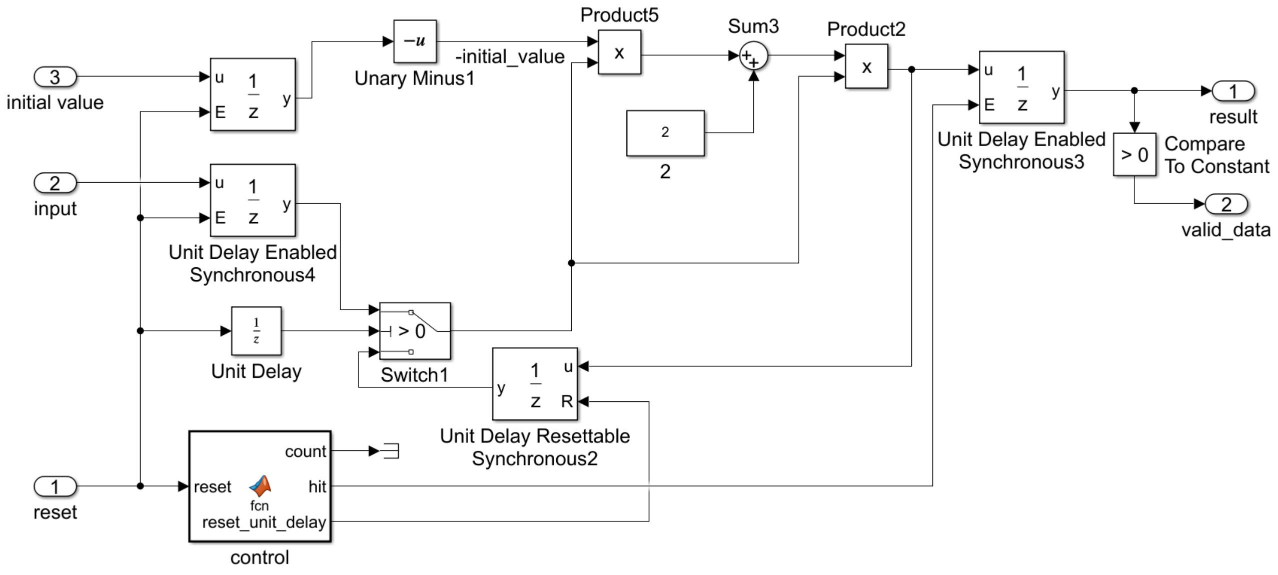


Figure 17: *reciprocal_newton_raphson_horner* block

The output result is shifted with the appropriate amount of bits in the right direction and is stored into an output unit delay enable block. If no window length is specified, then there is a default value of 0.005 which corresponds to 200 samples. This precaution is also implemented in the *sample_count* block to fit the default reciprocal value.

Power Average block

In the top level of this module, the input signals are multiplied and stored in the accumulator in its accumulative unit delay. The top level design of this block can be seen in figure 18.

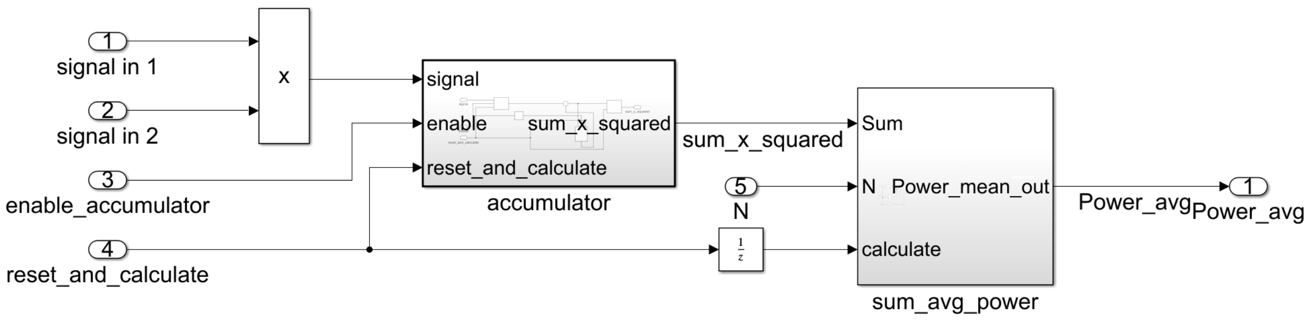


Figure 18: Power Average top

The input signals are implemented in $S17.16$, and the output signal of the cumulative sum is of size $S36.20$. This gives a sufficient amount of precision in the fractional part and also assures that no overflow occurs in the cumulative sum. When the accumulator receives logical one on the *reset_and_calculate* signal, the stored sum is passed to the *sum_avg_power* block, which computes the average power over the N specified samples by multiplying the reciprocal value of the window length and the accumulator is restored to zero. The *sum_avg_power* can be found in figure 19.

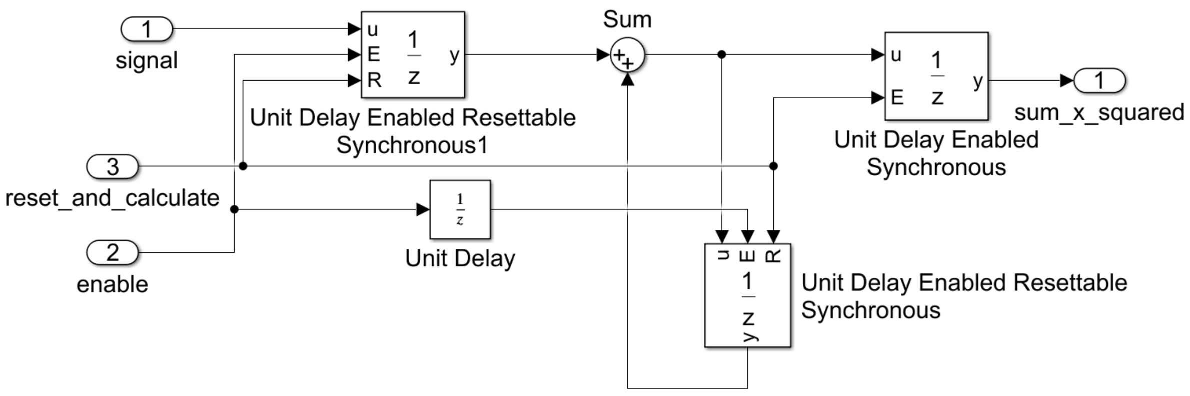

1
signal
3
reset_and_calculate
2
enable
u
E
R
1/z
y
Unit Delay Enabled Resettable Synchronous1
Sum
1/z
Unit Delay
u
E
1/z
y
Unit Delay Enabled Synchronous
1
sum_x_squared
u E R
1/z
y
Unit Delay Enabled Resettable Synchronous

Figure 19: *power_accumulator* block scheme

Effective value block

Top level

This block is responsible for the effective value computation. The top level is very similar to the top level of the *window_calculation* block. Again the *accumulator* stores the input signal value in its accumulative unit delay. The *accumulator* can be seen in figure 20.

However, in comparison with the *Power_avg* block, here only one signal is present and the input signal value is squared inside the *accumulator* since effective value is computed. The input signal is implemented in $S17.16$, and the output signal of the cumulative sum is of size $U32.20$.

Again, this gives a sufficient amount of precision in the fractional part and also assures that no overflow occurs in the cumulative sum. At the assertion of *reset_and_calculate* signal, the accumulator passes the cumulated sum to the *rms_core* block and resets to zero.

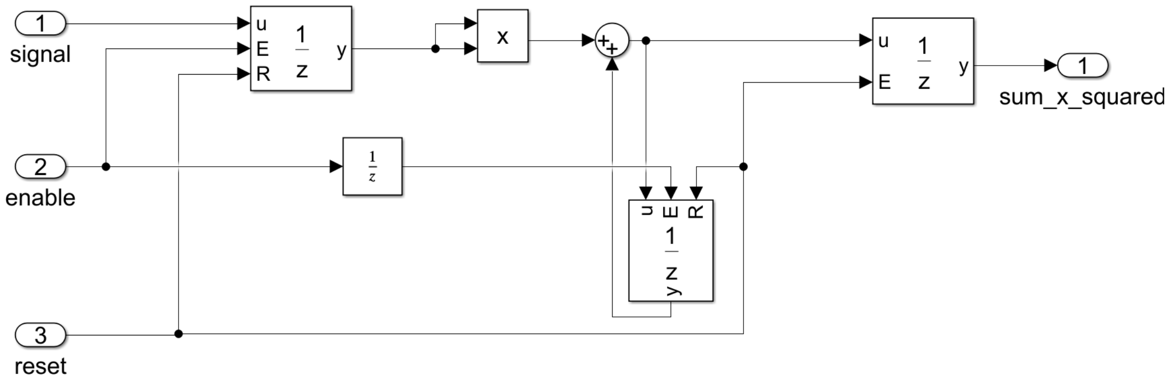


Figure 20: *RMS_accumulator* block scheme

RMS Core submodule

The *RMS_Core* submodule is the main block, which computes the effective value. It's top level is in figure 21. The operation process in this block is controlled by the finite state machine *rms_core_control*. If the *calculate* flag is set, the state machine starts. First it stores the accumulated sum and the reciprocal value in the unit delay enabled blocks and multiplies them.

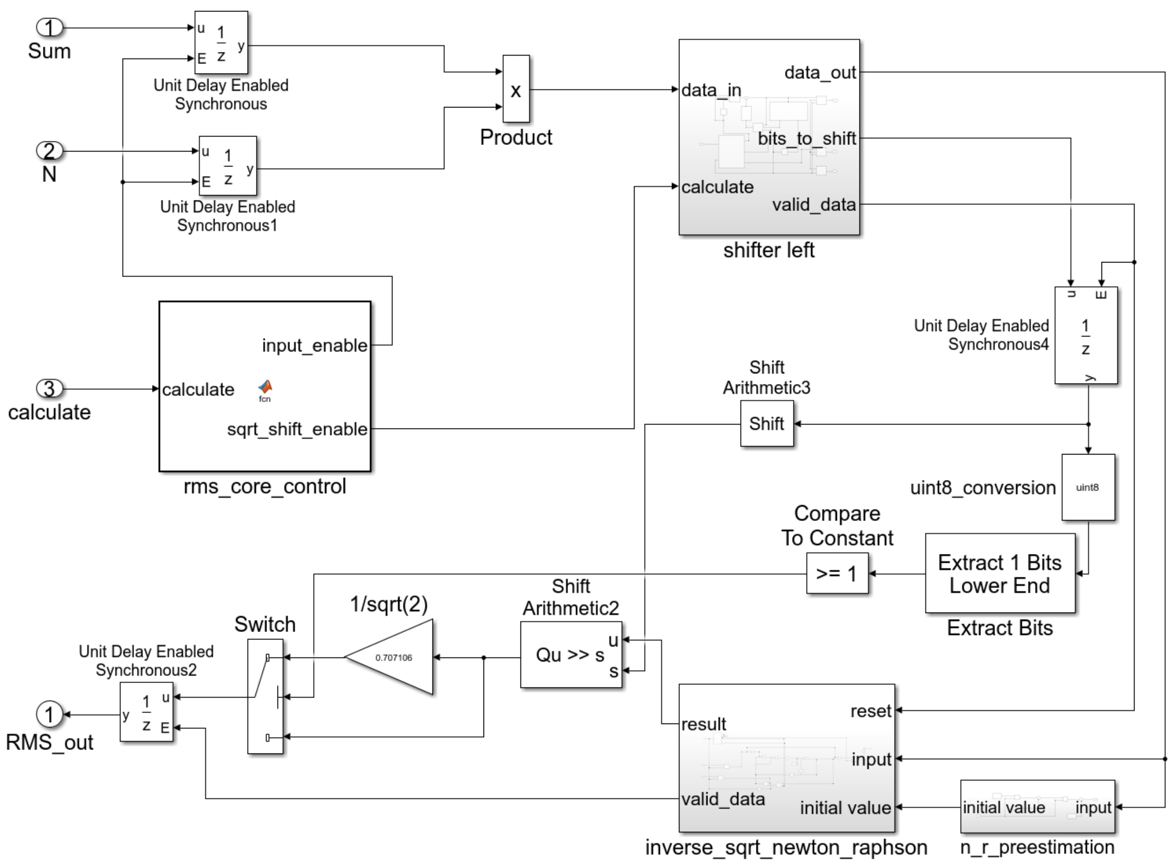


Figure 21: RMS Core submodule top level

This result then proceeds to the *shifter left* submodule. This submodule is the same as in the *window_calculate* block and can be seen in figure 22 and is explained in section 6. Its purpose is to rescale the input data for the Inverse square root Newton Raphson method as described in 5, because chosen interval is $[0.5, 1)$.

This scaled data is passed to the submodule *n_r_preestimation* where the preestimation for the initial value is found. Then the *inverse_sqrt_newton_raphson* computes the square root value. These are the same blocks as in figures 8 and 9.

Since the input signal into the sensor is in the interval $[-1, 1)$, the total value in the accumulated sum multiplied with the reciprocal number for the corresponding window length will not overflow the interval $[-1, 1)$. However, the opposite situation may occur, when the input signal values are around zero, the sum multiplied with the window length is less than 0.5 in which the Newton Raphson algorithm does not work properly with the desired precision. Therefore, a left shift is introduced. Also since the square root changes the output interval as described in the section 5, a correction by a factor of $\sqrt{2^k}$ which is equal to $2^{\frac{k}{2}}$ for odd numbers and $\sqrt{2} \cdot 2^{\frac{k-1}{2}}$ for even, where k is the number of bits shifted in the initial prescaling section, needs to be performed. However, this time the number will be shifted right by the correction amount, because initially the input number was in the proper interval, or it was too small and therefore shifted left.

The logic for correction includes a right shift, which corresponds to $\frac{k}{2}$. With this value, the square root result is shifted right. Depending on if the initial shift was odd or even, additional multiplication by a factor of $\frac{1}{\sqrt{2}}$ is performed and the block *Extract* 1 *Bits Lower End* decides whether this multiplication is needed or only the shift is sufficient.

For a better explanation an example is provided. *Sum* is the accumulated sum of samples, and N is the reciprocal value of the window length.

$$Sum = 40.9998$$
$$N = 0.0049999$$
$$=> result = Sum \cdot N = 0.204964$$

This result is shifted left two times i.e.

$$shift\_result = 0.204964 \cdot 2^2 = 0.81985.$$

This value is then squared, which gives the result 0.90546. Then shift with the corrected bit value is applied, here by one bit right which results into value 0.45273, which is the correct result of $\sqrt{0.20496}$ with a precision of 17 bits.

Shifter left **submodule**

The shifting process is controlled by the *shifter control* state machine. Whenever *calculate* flag is set, the input number is binary shifted on each clock cycle by one bit left until the input number is equal or less than 1. Then it passes the new shifted number to the output buffer and also information about the number of shifted bits. The *shifter* block's input is *uint*16 and the output is the standard fixed point value at which the *RMS Core* block operates *S*32.28.

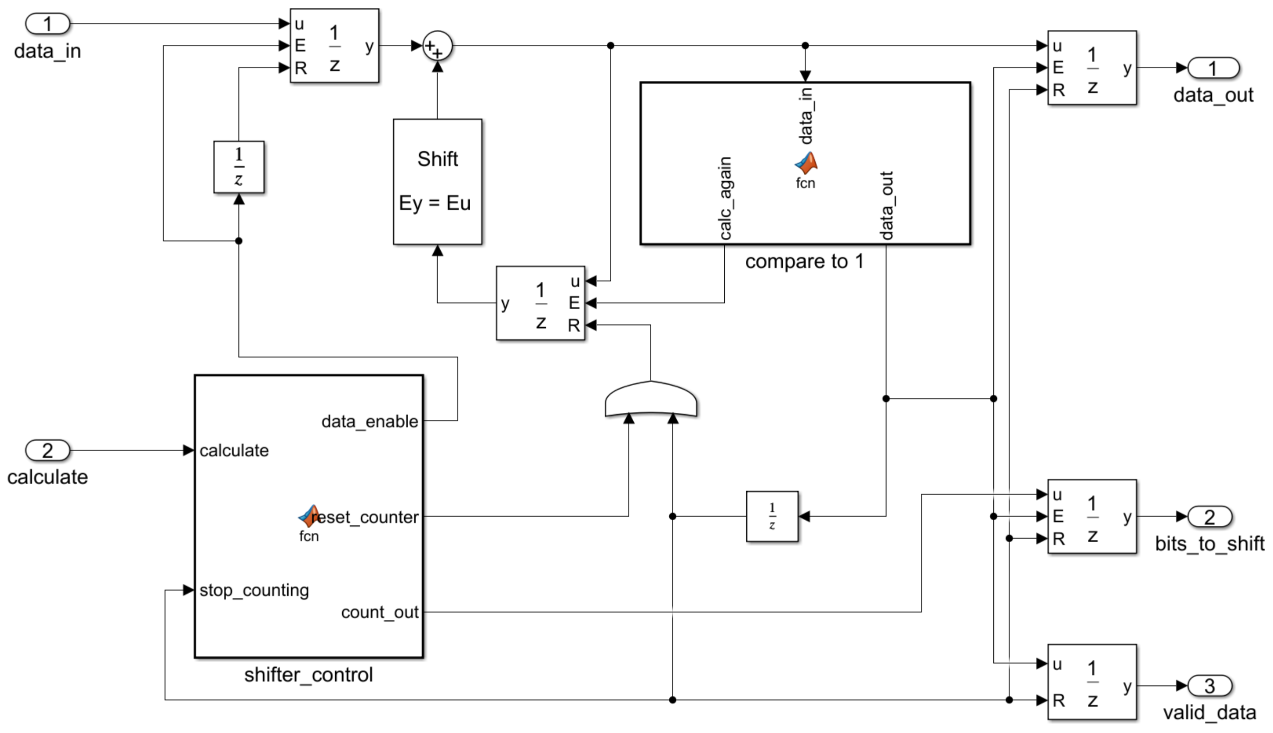


Figure 22: Shifter implementation

Apparent Power

For the apparent power, only the RMS values of calculated current and voltage are multiplied as shown in equation 5. The Apparent Power is calculated over the same window length as the Average power, because there is only one sample window length for the whole design.

Simulation results

Comparison of Power values simulated with the model and computed with an ideal moving average is in figure 38 in Appendix. Comparison of RMS values is in the next figure in the Appendix 39. Input signal is a sine wave of frequency 1000 Hz mixed with noise and sampled with sample frequency $f_s = 200\ kHz$. The mismatch of the signals at the state transition is due to the method of calculation of the ideal moving average. It calculates it sample by sample in contrast with this thesis design, which calculates it over an entire window as mentioned several times above. The whole model with the simulation sources can be seen in figure 23. The *signals_from_adc* block reads input signals generated from Matlab. The Constant block defines the window length and the *unitdelay* block labeled with *calc_enable* generates one time stimulus for the block *window_calculation* to recalculate the window reciprocal value.

Result Table	Difference (-)	Precision (bits)
Power Avg	3.836872e-05	14.66
Power Apparent	4.436492e-05	14.46
RMS I	3.540049e-05	14.78
RMS V	5.214707e-05	14.22

Table 2: Comparison Table of simulated model and ideal values, *Avg* stands for Average, *RMS* for Root Mean Square

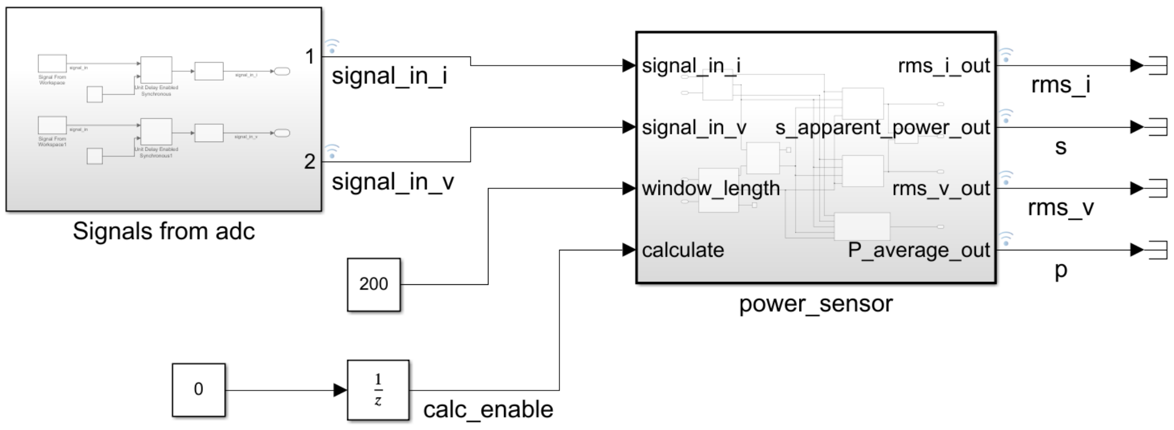


Figure 23: *Simulation model* block scheme

7 Introduction to Zedboard

Brief overview of the Zedboard and its capabilities

Zedboard stands for Zynq Evaluation and Development board [29]. Zynq is a platform with a SoC - System on a Chip which consists of a ARM Cortex - A9 processor and 28nm programmable logic from the Xilinx 7-series FPGA architecture [29]. The main device on the Zedboard is the XC7Z020 Xilinx chip with a Dual core ARM Cortex-A9 processor, external memory DDR3 support, lots of peripherals and the AXI standard interconnect [30]. The PL - programmable logic contains 85k Programmable Logic Cells, 53200 Look-Up Tables (LUTs) and 106400 Flip Flops. It also contains 140 Block RAMs of 36Kb and 220 DSP Slices with a width of 18 bits. Because of the two independent parts, the PS - processing system and the PL, custom logic and custom software may be implemented, where either the processor would be too slow or the design of the programmable logic too complicated. Usually, the PL is used for a high speed logic and signal processing, whereas PS supports software routines and operating systems. These two systems are linked together via the industry standard Advanced eXtensible Interface (AXI). In the chip, there is also an AD converter called XADC which is provided to the design environment in the form of an IP - intellectual property package. The XADC is a dual 12-bit 1 Mega sample per second analog to digital converter. It features access up to 17 external analog input channels and also to several on chip sensors such as the die temperature and on chip power supply monitoring sensors [31].

Several other functionalities are present on the Zedboard, such as 512MB DDR3 memory, audio inputs with a audio codec, Ethernet and JTAG connectivity, HDMI and VGA output for image and video processing, OLED display, switches and push buttons as well as LEDs. There is also a SD card slot which allows booting of custom Linux image onto the PS or serves as a non-volatile external memory [32]. On the PS usually runs an operating system, which is responsible for data exchange between the platform and a host device over JTAG or Ethernet interfaces [33].

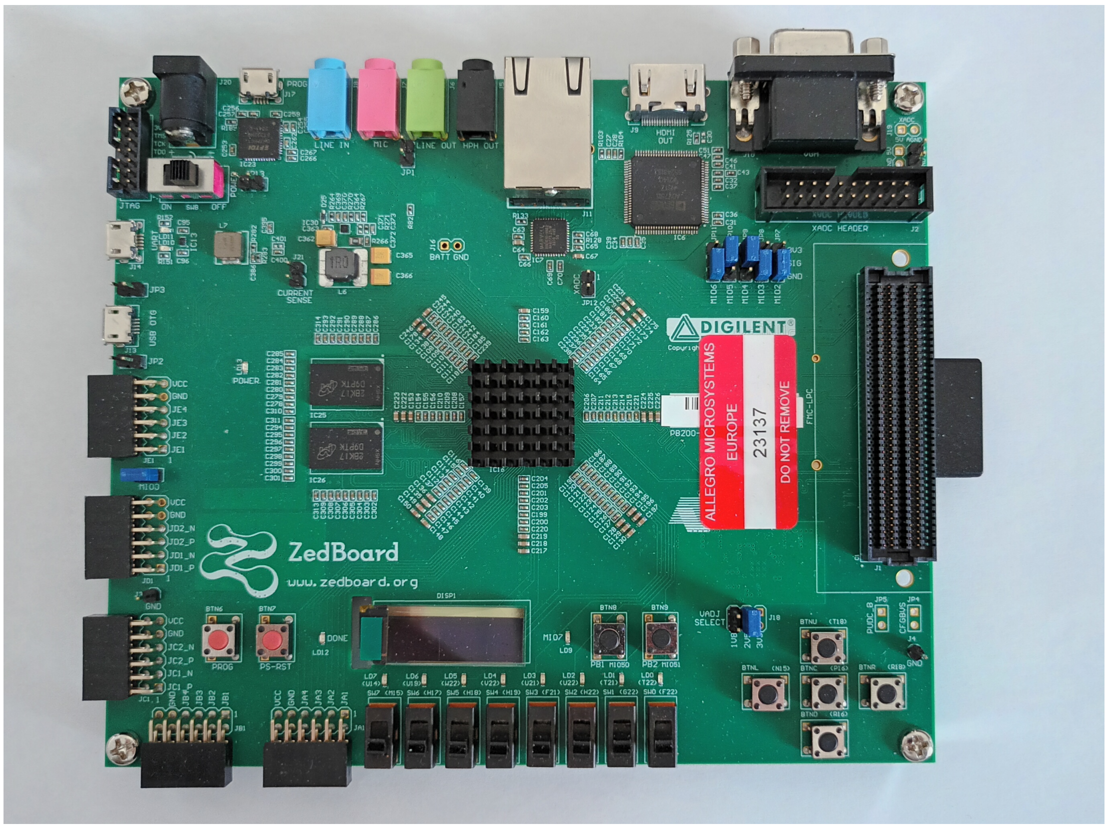


Figure 24: Zedboard evaluation board

Preparation of Zedboard and Matlab for FIL and External simulation

For establishing a connection between the Zedboard and Matlab Simulink, several ad on packages are required, amongst the HDL Coder and Verifier also HDL Coder Support Package for Xilinx Zynq Platform and an Embedded Coder Support package for Xilinx Zynq Platform [34].

Next using these packages, a Linux Image must be loaded onto an SD card from which the embedded platform will boot. Then using the proper jumper settings, the boot location from SD card is chosen. Also, the host device (computer with Matlab) needs to be in the same IP address space, here for example 192.168.1.11 for the host network card and 192.168.1.10 for the Zynq platform [35]. Connection over JTAG is also possible. Once the Linux is running on the Arm, it is possible to connect to the board using a Serial communication or over SSH via ethernet. When connecting over ethernet a username and password must be first inserted. If the image and communication work properly, the Zedboard should respond to basic Linux commands and for example a folder structure can be observed. Then from Matlab an object is constructed calling *zynq* function [36] which creates a connection to the development platform. Connection via a micro usb for bitstream programming is also needed, as well as the power supply.

8 Design Validation on FPGA

Using FPGA in the Loop it is possible to validate the design directly from Matlab. Matlab package HDL Coder and HDL Verifier can regenerate the designed Simulink model into several Hardware Description Languages (HDL) files and pack it into an independent IP core, which corresponds with the model implemented in Simulink. With HDL Workflow Advisor [8], the model is generated for FPGA implementation. Several settings need to be configured in the workflow, including proper board selection - Zedboard, synthesis tool - Xilnix Vivado, setting target frequency - this design 10 MHz, how does the HDL Coder translate the model to HDL code. After these settings, Matlab generates the Vivado project, includes generated Verilog files, adds it's JTAG controller and a clock wizard IP from Xilnix and generates a Wrapper file which acts as the top level file.

Then the Vivado Design Suite is issued as an external tool which handles the required steps for bitstream generation. These steps include RTL Analysis which translates the algorithmically written HDL files and model into the register transfer level code of needed hardware blocks. The next step is the logic synthesis, which generates from the RTL the netlist and optimizes the used hardware resources and critical paths in the design. Then the Implementation of the netlist is performed, where physically present hardware blocks including Programmable logic cells, LUTs, DSP slices and Flip flops are chosen and nearest paths are examined, found and optimized. Place and route then places the block and routes the connections between them and performs a Static Timing Analysis (STA), where it check for example if the signal in the critical paths arrived in proper manner, or performs clock jitter analysis [37, 38].

A new model is also generated in Matlab Simulink, which can be seen in figure 25. The *power_sensor_fil* block executes the hardware component on FPGA board. The FPGA is programmed from this block with the generated bitstream, and signal representation may be changed in this block as well. Simulation is run both in the model *power_sensor* in Simulink and in the generated IP implemented in the FPGA. Then the difference is compared in the Compare block. The input signals are generated in Matlab similarly as in section 6. In the *Compare* subsystem, each output signal is compared with the output from the Simulink model and if a difference occurs, a flag is asserted by a *AssertEq* submodule. This submodule can be seen in figure 42 in the Appendix. The proposed design tested without any mismatch against the Simulink model. However, due to few large critical paths, the design works only up to a frequency of 10 MHz. This issue is addressed in later sections 9. The design runs in the PL - Programmable Logic part of the board and communicates and sends data between the host and FPGA over JTAG interface.

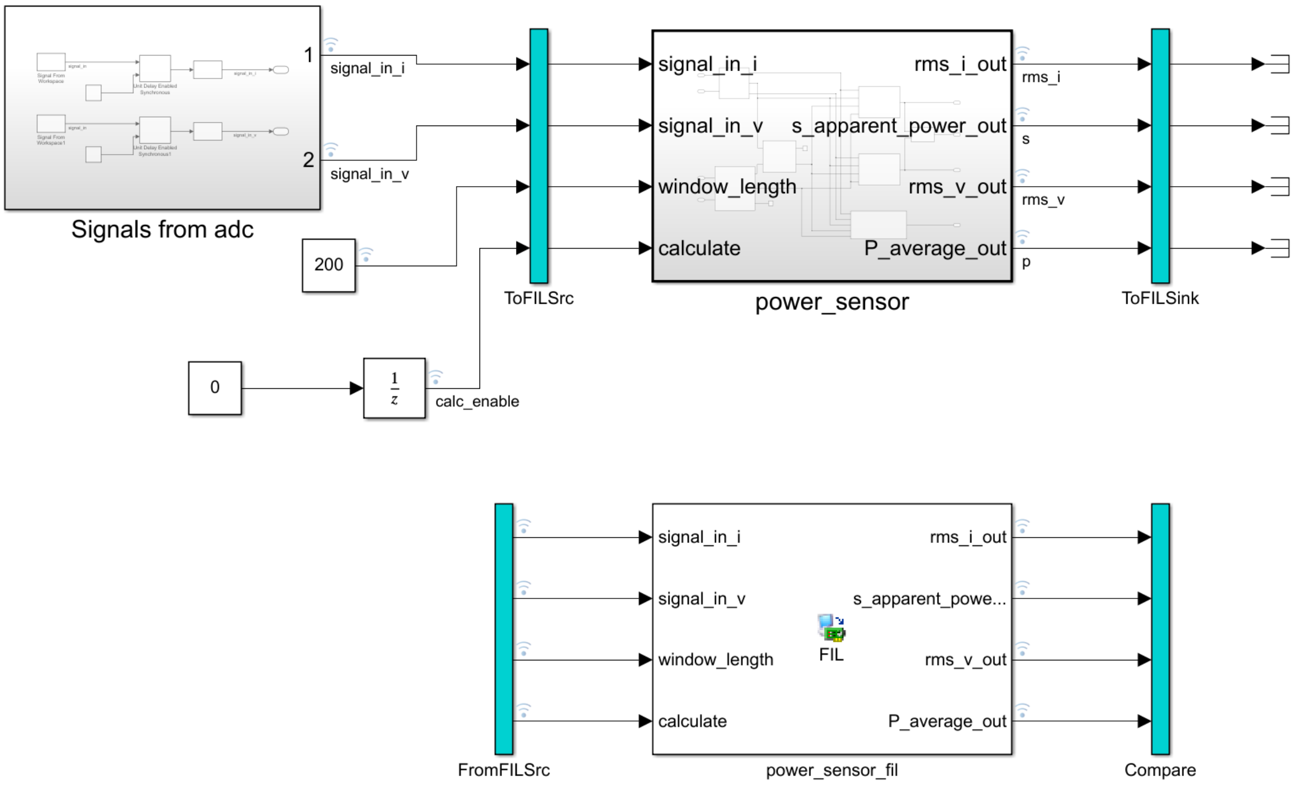


Figure 25: FPGA in the Loop Simulation

9 FPGA implementation

To obtain analog signals for the power sensor, current would be measured for example with hall plates or a shunt resistor [3] and converted to voltage. These analog signals would then be transformed to digital signals using an AD converter. As stated above in section 7, the Xilinx FPGA chip contains an AD converter module named XADC, which can be instantiated and reconfigured using an Intellectual Property block in the Vivado Design Suite [31]. Signals for the purpose of emulating current and voltage are generated using a STM32F303RE Nucleo board and sampled with this ADC.

XADC introduction

Brief information about the XADC is provided in section 7. Inside the XADC, there are two independent converters, ADC A and ADC B. They can operate in several modes, from which Simultaneous Selection with Continuous Mode have been chosen. Simultaneous Selection means that the sequencer of each ADC automatically switches between selected input channels which were enabled when instantiating the IP, so ADC A between Auxiliary channel 0 -7 and ADC B between channels 8 - 15. This setting helps to maintain the appropriate phase difference between measured signals. Several sensors on chip can also be accessed through this sequencer. In Continuous Mode both ADCs sample, hold and convert the input signal values automatically without external triggering event [31].

Converted values from AD channels as well as sensor data are stored in dedicated registers. In similar registers, configuration data for all settings configurable in the IP are also stored. These data can be accessed in several ways. Besides using the AXI – Lite interface, there is also a FPGA interconnect called dynamic reconfiguration port (DRP), or JTAG. The DRP is a 16 bit synchronous read and write port. XADC might also output the data as an AXI4Stream and using Direct Memory Access (DMA) store them directly in a memory location such as the DDR3. ADCs can also be reconfigured while running using an AXI-Lite interface or directly from the PL using a DRP interface [31].

External Analog inputs are differential to reduce common mode noise on the external analog signal. Two connection pins are needed for one channel, Vpositive and Vnegative input, and the ADC samples the signal with the common mode, whereas it removes it and rejects noise. Unipolar or Bipolar mode can be chosen, with nominal input range form $0\ V$ to $1\ V$. The LSB size is $244\ \mu V$ producing zero code *0h* when $0V$ signal is present and *FFFh* when $1\ V$ in Unipolar mode. In Bipolar mode, the LSB value is the same, however the input voltage is in range $-0.5\ V$ to $0.5\ V$ and ADC are producing output in Two's Complement Coding [31].

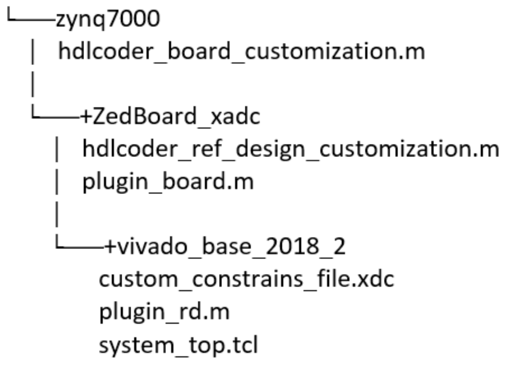
Zynq SoC Implementation Workflow

Workflow of implementing the power sensor on FPGA is explained in this section. It is similar to the workflow explained in section 8. Difference is in targeting the whole SoC platform, whereas in the Validation, only a block for JTAG communication with Matlab was added to the generated files from Simulink model. Now the model designed in Simulink is converted to Verilog and packed as an IP core. Matlab then takes the created IP and creates a Vivado project from a pre-defined Block design which contains the *Zynq7 Processing System*, *AXI Interconnect* IP, *Processor System Reset* and a *Clocking Wizard*. Into this block design, the generated IP is inserted and defined port connections are connected. Then a Verilog wrapper is created and Vivado runs the same synthesis and implementation process as described previously. After that the bitstream can be programmed into the FPGA either from Matlab HDL Coder or directly from Vivado Design Suite. After that usually a project for the Arm CPU is created and the processor is programmed with the created application. Matlab instead creates another Simulink model, where it swaps AXI ports previously defined in the HDL Coder Workflow Advisor with different subblocks, which drive the AXI communication. Then this newly generated Simulink model is compiled with the Embedded coder, connection to Zedboard is established as described in section 7 and the Arm is programmed with the created code. The last step is to run the Simulink model in external mode simulation. If there is communication between Matlab and the Zedboard set up, the results can be directly observed in Matlab.

However this predefined block design, which is used as template creation for the Vivado project needs to be adjusted to include the XADC IP block, to connect it with the overall block design of the Zynq SoC, define correct interface connectivity and especially explain Matlab to take into account these changes, so they are usable in the HDL Coder Workflow Advisor and that the proper code generation may be executed.

Registering custom reference design

The process of creating a new project used by the HDL Coder Workflow Advisor is called registering custom reference design. This includes the Vivado block design, as well as several scripts used by Matlab. It needs to follow a certain folder structure, best explained in this work [33] or on the Mathworks webpage [39]. This folder structure itself, explained with a diagram in figure 26, needs to be located between other support packages and reference designs. First script called *hdlcoder_board_customization.m* is the board plugin registration file, which registers the folder structure and its object *plugin_board*. In the similarly called script inside the folder structure, the board object of HDL Coder is created, with the proper board name which appears in the Advisor, vendor, family, device, and package. Several other settings are also defined such as JTAG chain position and external Input Output interfaces. The folder structure may consist of several reference designs of the same board with various Vivado block designs. Script called *hdl_coder_ref_design_customization.m* takes this into account, the correct board name needs to be set and all desired reference design definitions are registered here via corresponding *plugin_rd* located in corresponding subdirectories. These also contain the Vivado block design in the form of a *.tcl* file, which stores the commands for the Vivado tcl console and creates a Vivado project and executes the referenced block design upon selecting it in the HDL Coder Workflow Advisor.



```
└──zynq7000
  | hdlcoder_board_customization.m
  |
  └──+ZedBoard_xadc
     | hdlcoder_ref_design_customization.m
     | plugin_board.m
     |
     └──+vivado_base_2018_2
        custom_constrains_file.xdc
        plugin_rd.m
        system_top.tcl
```

Figure 26: Folder structure for Registering custom reference design

The script *plugin_rd* is responsible for defining interconnection inside the block design. Reference project object is constructed with the correct board name and corresponding custom block design. Then a clock interface is added which matches the exact block already created in the Vivado block design, with adjustable clock frequency but fixed wiring between other peripherals. Same name - property *ClockModuleInstance* as in the block design must be set in the script, as well as other connections, like *ResetConnection* which is connected to the *FCLK_RESET_N* on the *Zynq7 Processing System* block. Frequency is set inside the Workflow Advisor and appears afterwards in the Vivado project.

An *Axi Interconnect* is added, which connects the generated IP package with the *Zynq7 Processing System* block through AXI4Lite interface. Connection between the processor and this block is defined in the block design, however the connection between the generated IP and the Master interface on the *Axi Interconnect* block are here defined. Also correct Master address space corresponding to the address in *Zynq7 Processing System* is set. The last interface connections with the instantiated XADC IP block are defined, which is created in the Vivado block design using the *IP Integrator*. Output Bus of a 16 *bit* width where data are transferred from the XADC registers. One output signal called *drdy_out* is also present, which signalizes if the data to be red are prepared. For the input, one address bus of 7 *bit* width with a 1 *bit* signal is responsible for addressing registers to be read. Input port *di_in* can be used for reconfiguring setting of the XADC, however for the power sensor XADC does not need to be reconfigured during execution. Identical opposite connection ports need to be assigned via the Workflow Advisor to output or input ports of designed model with matching communication protocol implemented. In figure 27 is the block design with already instantiated XADC and inserted and connected power sensor IP. This is the result of the HDL Coder Workflow Advisor, which is synthesized and implemented into bitstream. The block design exported for the Workflow excludes the block *power_sen_ip*.

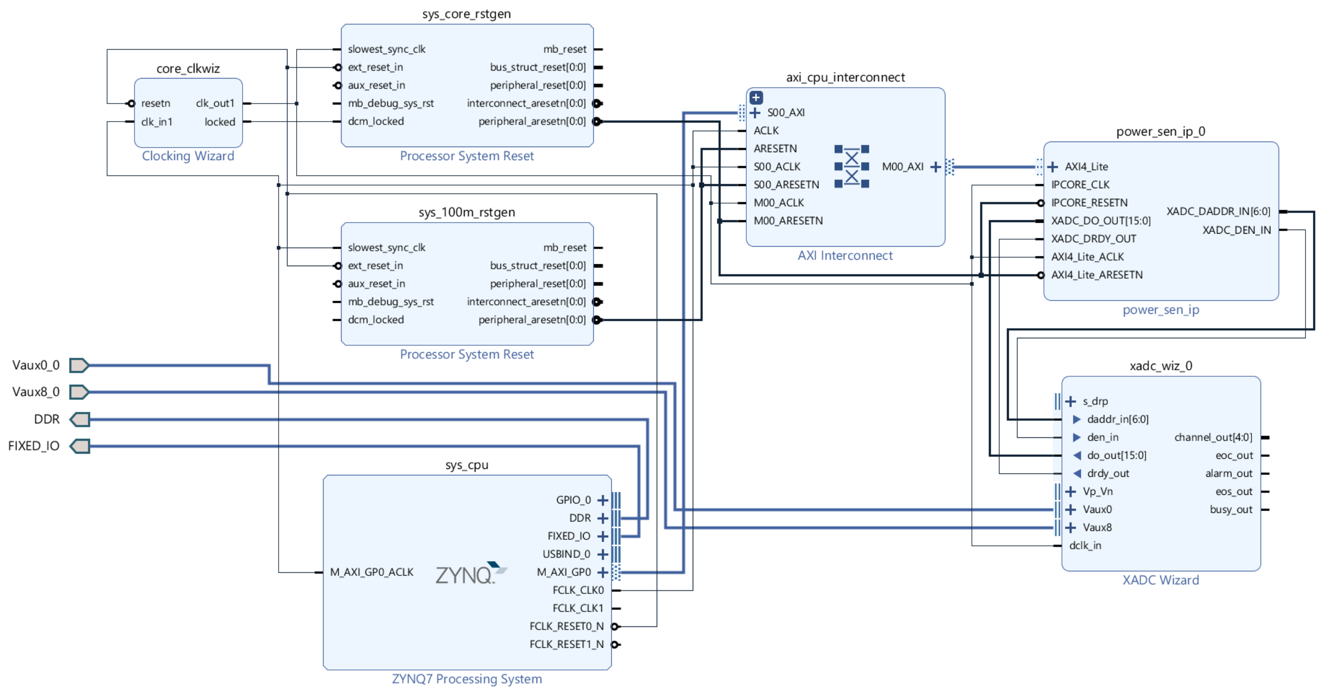


Figure 27: Vivado block design

XADC instantiating

For the power sensor, two channels Auxiliary channel 0 - Aux0 and a corresponding opposite channel 8 – Aux8, were assigned to the ADC A module and to ADC B module respectively in Unipolar mode. Channel sequencer with simultaneous Selection has been configured, as well as Continuous Mode. Frequency of the main clock is set to 10 MHz, which is similar to the main clock frequency as the rest of the design. ADC conversion rate is set to 40 $KSPS$ (kilo samples per second), so the acquisition time is equal to 4 clock cycles of the XADC. No averaging is selected, and no alarms measured with the internal sensors. The used interface is the DRP.

XADC interface

According to the XADC user guide[31], the DRP interface is best suitable for implementation with a state machine. In figure 28, the timing diagram overtaken from [31]is shown of a read and write operation.

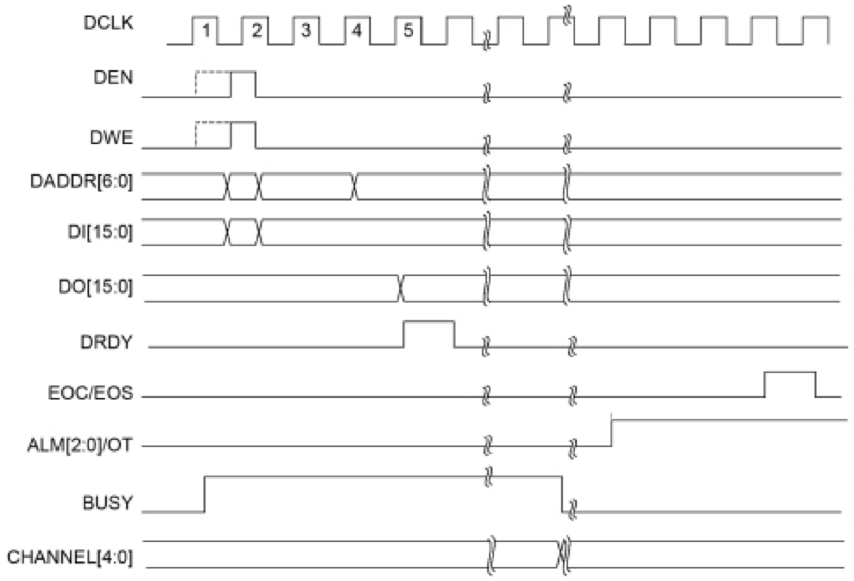


Figure 28: DRP timing diagram.

To read out converted data stored in the result registers, first the data enable DEN needs to be asserted to logic 1 for a clock cycle. At the next clock cycle, the DRP address $DADDR$ is captured. If DWE signal is low, then a read operation is executed, and corresponding data is put out on the data bus DO - Data Out. If the data are prepared, signal $drdy$ changes to logic 1 for a clock cycle [31]. Then the state machine reads the data from the output port. Since two registers need to be red out, the state machine changes its address index to the second address and performs another read. Then it changes the address index back to the first one. This way it cycles and reads out continually these two values.

In figure 29, the state machine with additional demux and two output flip flops can be seen.

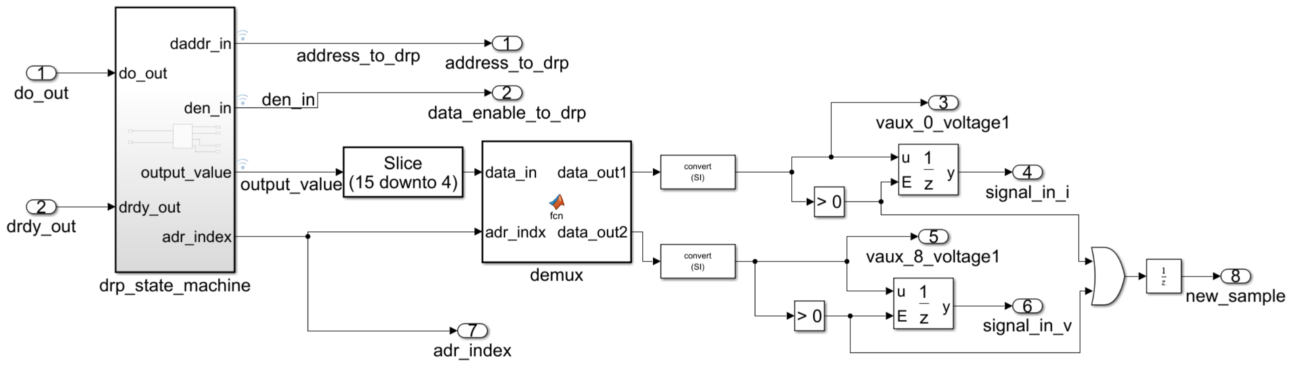


Figure 29: DRP submodule [31].

The submodule *drp_state_machine* works as described above. The inputs are *do_out* and *drdy_out*. Both are connected to ground on the top level of the design in order to rightfully assign them in the Workflow Advisor. Port *drdy_out* serves as the data ready signal from the DRP interface of the XADC, *do_out* is the data bus of 16 bit width, however, only 12 bits are valid and bottom bits are of value 0 at each data transfer [31]. Port *address_to_drp* is the 7 bit wide address port and *den_in* is the data enable signal. These are connected to proper interconnections via the Workflow Advisor and thus through the *plugin_rd* script. The output values need to be sliced down to 12 bits and then are demuxed (separated between each other) using the *demux* function either to voltage or current, depending on the address index *addr_index*. Otherwise, they are equal to 0. If the value is nonzero, then the flip flop (Unit enabled synchronous) is turned on and a new value is put into the output. Also, a *new_sample* signal is generated for the other modules in the top level of the design.

Top level of Implemented design

The top level is in figure 30. The computational core is the same as in the section *Proposed power sensor architecture*6. However, the submodule *new_sample* is removed and instead for its function the submodule *xadc_communication* is responsible. It produces the same outputs for the core as the old submodel, but few additional outputs are present.

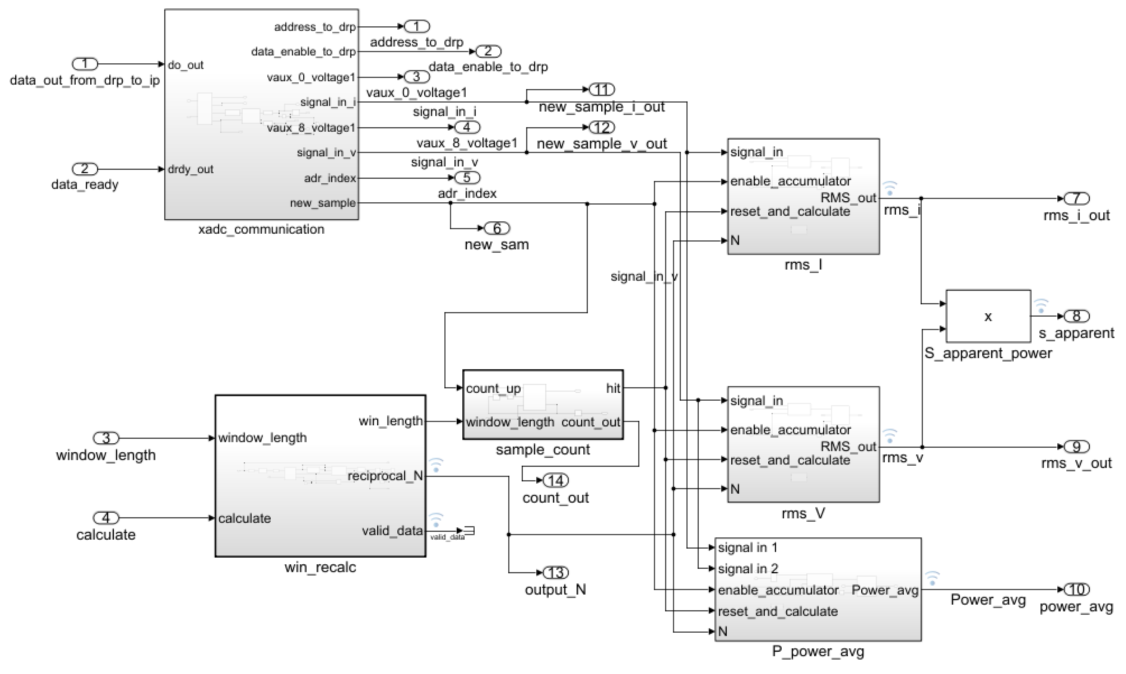


Figure 30: Top level of Implemented design

Difference also occurs in the *reciprocal_newton_raphson_horner* block present in the *P_power_avg* submodule, and the equivalent submodules which calculate using the Newton Raphson method located in the RMS submodules (*inverse_sqrt_newton_raphson*). These blocks which are responsible for the finding the reciprocal value or the square root have long combinational paths which are poorly implemented by the synthesis process in Vivado [38]. If the frequency is larger than 10 MHz, error in the worst slack occurs, which means, the combinational path between two flip flops is too long and the signal does not arrive in the proper manner within one clock cycle. To resolve this issue, additional pipelinening might be introduced [40]. I resolved this by manually adding a *unit_delay* block inside the design between the two multiplexers and adjusting the control block, which controls the inner reusing of values for calculation, so it is delayed by the correct amount of clock cycles. This increased twice the time it takes to calculate the result value, from 2 clock cycles to 4, however the speed of the design increased to 20 MHz, which is a large improvement.

The changed design of the cores can be seen in figure 31 and in figure 32.

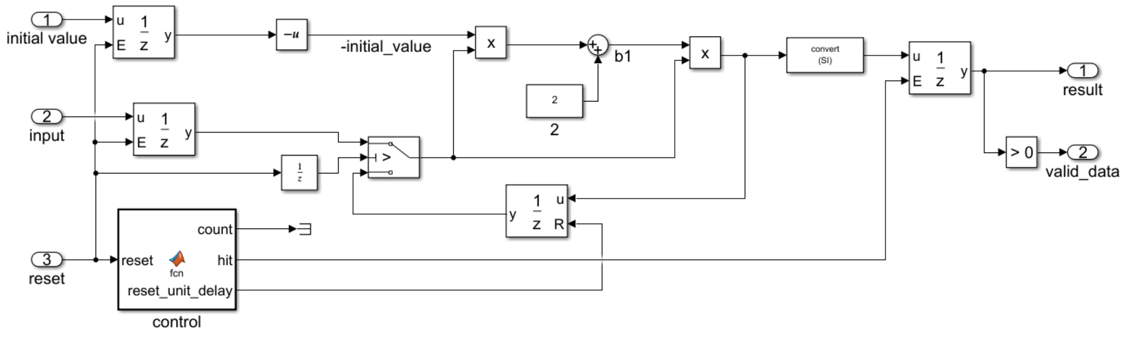


Figure 31: Modified *reciprocal_newton_raphson_horner* block 17

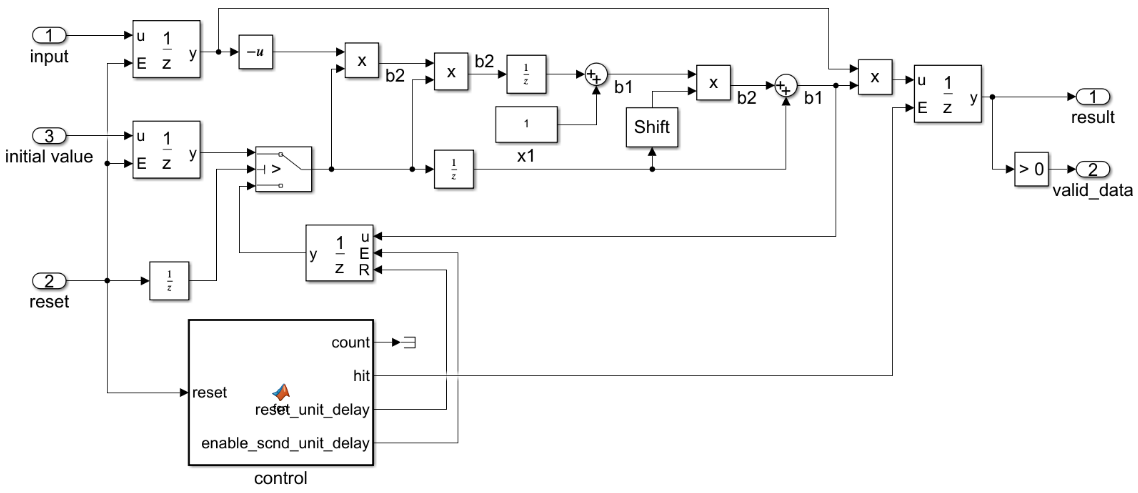


Figure 32: Modified *inverse_sqrt_newton_raphson* block from 8

The difference between the timing is in table 4. These values are obtained from Vivado from the Implementation step. In table 3 is the difference between used hardware resources of these two designs.

Used Hardware resources	10 MHz Design	20 MHz Design	Difference
LUT Synthesis	3355	3366	11
LUT Implementation	3059	3055	4
FF Synthesis	3409	3505	96
FF Implementation	3157	3253	96
DSP Slices	47	47	0

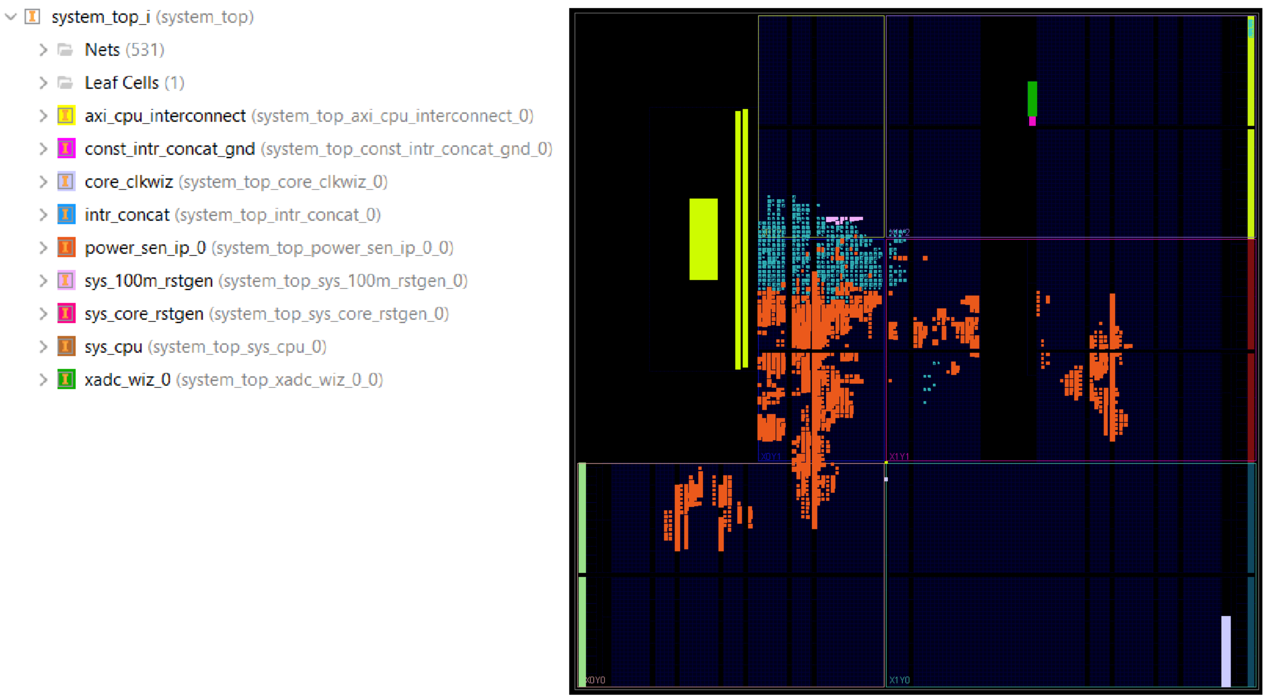
Table 3: Comparison of Used Hardware Resources, *LUT* stands for Look Up Tables, *FF* for Flip Flops and *DSP* for digital signal processor.

Frequency of design	10 MHz Design	20 MHz Design
Setup - Worst Negative Slack (WNS)	3.764 ns	4.948 ns
Hold - Worst Hold Slack (WHS)	0.035 ns	0.036 ns
Worst Pulse Width Slack (WPWS)	3 ns	3 ns

Table 4: Comparison of Design Timing Summary

The higher speed design (20 MHz) has more resources used in the model than the 10 MHz design. Difference can be seen in both Flip Flops and LUTs, Flip Flops differ exactly by the number of added Flip Flops in the design, which is $4 \cdot 24$, where 4 is the number of added *unit delay* blocks (FFs) and 24 is the bit width of the processed signal. Lower speed design (10 MHz) uses slightly more LUTs. This is probably due to different solutions of the synthesis and implementation optimalization algorithms. DSP Slices do not differ, as these are large blocks and the same amount of multiplexers is used.

In figure 33 is for ilustration the implemented design in Vivado with colorfully highlighted implemented parts. The designed IP module is in orange. XADC as green with the interconnections as pink can be seen in the right upper corner.



system_top_i (system_top)
- Nets (531)
- Leaf Cells (1)
- axi_cpu_interconnect (system_top_axi_cpu_interconnect_0)
- const_intr_concat_gnd (system_top_const_intr_concat_gnd_0)
- core_clkwiz (system_top_core_clkwiz_0)
- intr_concat (system_top_intr_concat_0)
- power_sen_ip_0 (system_top_power_sen_ip_0_0)
- sys_100m_rstgen (system_top_sys_100m_rstgen_0)
- sys_core_rstgen (system_top_sys_core_rstgen_0)
- sys_cpu (system_top_sys_cpu_0)
- xadc_wiz_0 (system_top_xadc_wiz_0_0)

Figure 33: Implemented Design in Vivado

10 Measured results with implemented design

After the bitstream generation in Vivado, the generated Simulink model for programming the processor is opened. The design is replaced with an automatically generated subsystem, where previously assigned ports are exchanged for AXI4-Lite read or write interfaces, depending on if input or output was assigned. Other ports which were assigned as external ports are connected to ground, since they are not used by the Axi4-Lite communication but by the data transfer between the IP and DRP of XADC. The generated model is in figure 34. The generated top level of the model is in figure 48 in the Appendix. The submodule *AXI4SlaveRead* is also in the Appendix, in figure 49, and contains Simulink Embedded Coder library blocks with proper Register offset, Device name, and Data type for the *Zynq 7 Processing System* to access correctly the result registers. The only change required to perform is to set the sampling time to atleast 0.001 seconds. Then *Simulation in External Mode* is run and the model starts to transfer data over JTAG via the processor from the implemented IP model.

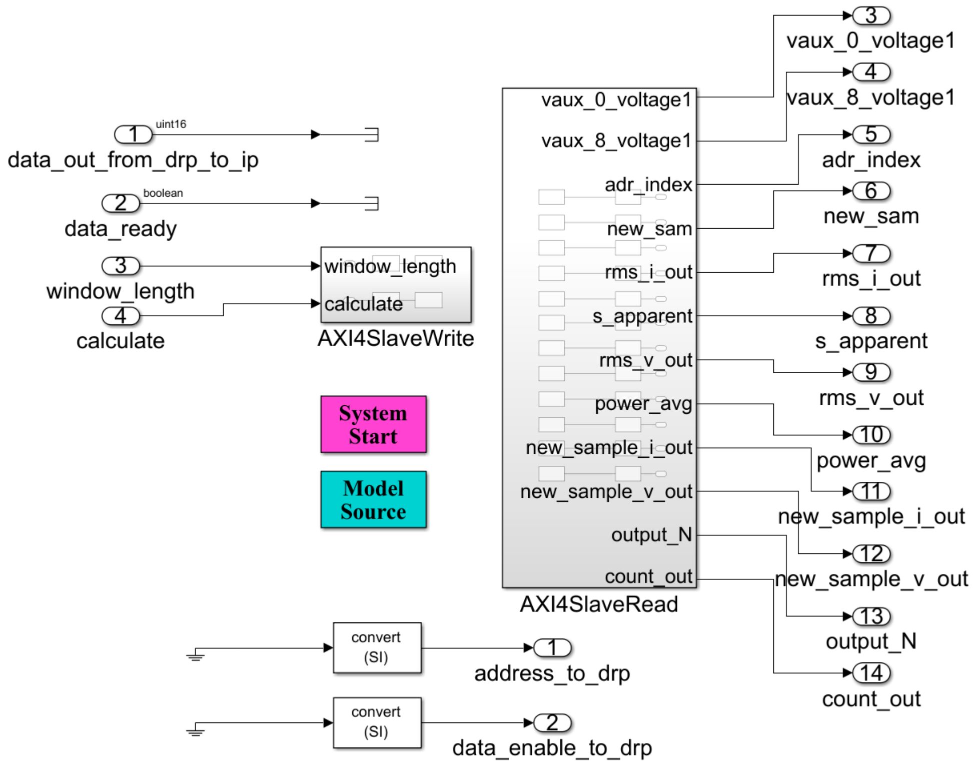


Figure 34: Automatically generated model for the Embedded Coder and measured data transfer.

In figure 35 Zedboard conntected to host computer and signal generator can be seen. For the signal generator, the STM32F303RE Nucleo board has been used. It features two 12-bit Digital to Analog Converter (DAC) channels with simultaneous conversion [41]. For demonstration purpose, two sine waves are periodicaly generated from the DAC and connceted to the input pins of the XADC on Zedboard. These sine waves have a different amplitude and phase shift, to demonstrate implemented functions. Four wires are conntected from the Nucleo board to the XADC Header. Two which are ground on the Nucleo board, are connected to the negative AUX 0 and AUX 8 input and two connections are outputs from the two DAC channels, pins D13 and A2 on the Nucleo board, connected to the positive AUX 0 and AUX 8 input. The Nucleo has been programmed with the STM Cube IDE [42].

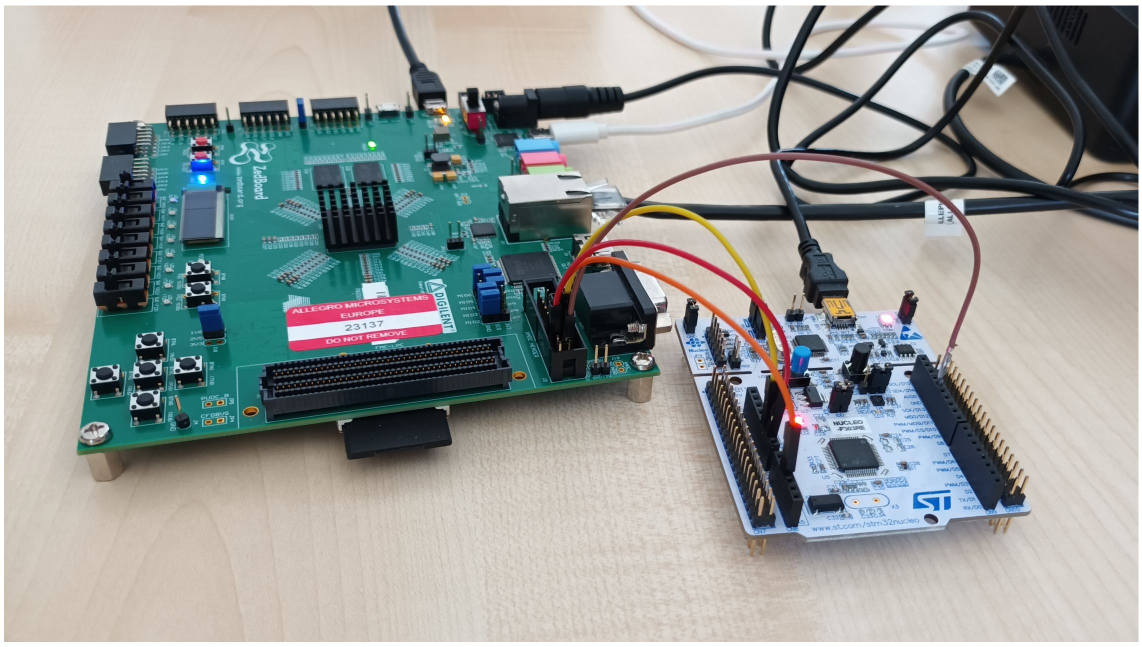


Figure 35: Zedboard connected to the signal generator and host pc.

In figure 36 the XADC header on the Zedboard can be seen, figure is overtaken from [43]. To this header, the Nucleo board is connected.

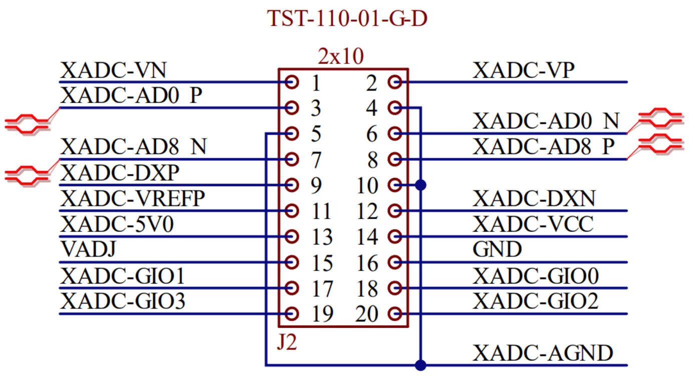


Figure 36: Zedboard XADC header schematic.

With the correctly set up Simulink model and bitstream programmed into the FPGA, the data transfer of measured data is started. Measured results are in figure 40 or in figure 41 in the Appendix. In the figure 40 partial waveform diagram of measured signal with two window length changes might be seen. At the first window length change, the window was changed to 10000 samples. A larger gap between new values of output signals is observed. Then a next window change to 600 samples was performed and results change more often, also an initial gap to lower values appears. Output values have larger ripple. This can be seen in the figure 37 which is a detailed diagram with window length of 600 samples. As overall measurement waveform plot serves figure 41. Result equal to zero at the begining might be seen, since the signal first needs to be stored before computation. Also, the window changes and corresponding ripple is visible in the same figure.

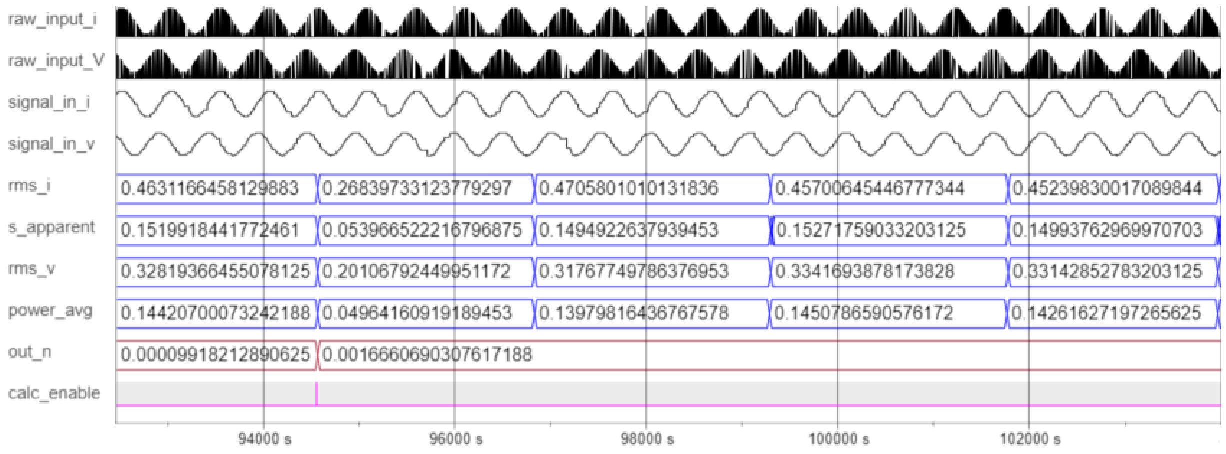


Figure 37: Measured result with window change detail

In the detailed plot in figure 37 the input signal might be seen with harmonic distortion. This is due to missing samples, which was poorly transferred over the JTAG interface, however for function demonstration, the output values, which persist over a greater sample size, transfer sufficiently

11 Further Improvements

Further improvements of the power sensor could be distinguished into several categories. For algorithms, improvements could be made in precision, speed or used hardware resources.

For the used hardware resources, the whole design could use only one computational core, which would calculate the reciprocal and square root functions, as well as proper shifting of all signals, and would store and pass these results to the output at the same time.

For the speed, for example when using the CORDIC algorithm, additional pipeline registers could be added, the core could be hardwired trough all stages, and input samples would be processed simultaneously, so after an initial delay, after each clock cycle would be a new value at the output.

Improvements with precision could be achieved with more precise signal sampling or more computation runs of the algorithms.

For the FPGA implementation and demonstration of the power sensor, a better way to transfer data between the evaluation board and Simulink could be found, for example using ethernet connection which is far superior with data transfer than the JTAG interface. Or perhaps to create a bus interface, that would send data stored in memory, however these features are out of scope of this thesis.

12 Conclusion

All goals defined in the Assignment have been successfully fulfilled.

The purpose of this thesis was to conduct research about algorithms used in ASICs or FPGAs for fast, accurate, and low hardware resource requirement elementary function evaluation, ultimately resulting in designing a power sensor. Model based design aproach has been used and FPGA in the loop for rapid development. For the first Assignment objective - Research on the topic of fast computational algorithms for specific functions needed for the signal processing core of a power sensor, a theoretical background about power, Model Based Design approach used throughout the whole designing process as well as fixed point numerical representation has been studied.

For the second thesis objective - Modeling of each method and comparison of accuracy, speed, and used hardware resources, several algorithms have been studied.

Research and derivation of equations has been conducted. Then a test function – square root has been chosen and researched knowledge has been applied to design a computational model in Simulink. Results have been simulated, from which precision and speed have been determined. Furthermore, these models have been generated to Verilog - a Hardware Description Language, Synthesized and Implemented for a FPGA device, from which hardware resource usage has been determined. All these algorithms have been compared and the most fitting has been chosen for implementation in a power sensor.

Proposed power sensor architecture has been created in Simulink which included computation of average power, apparent power, and effective values of both measured current and voltage. Behavior was simulated, and precision has been compared with ideal values calculated with Matlab, where the computational core reached a precision of 14 bits, also partly given by the chosen fixed point numerical implementation.

For the last two thesis objectives - Implementation of designed core and validation on a FPGA using FPGA in the Loop and Simulink, and also - Implemented functions demonstration - measurement and testing with real signals, a brief introduction to the FPGA evaluation board - Zedboard, connection to Matlab, and validation on the FPGA has been shown.

Then the Zynq SoC Workflow has been explained, with registering a custom reference design, instantiating the XADC IP module in Vivado and successfully generated bitstream. Issues with critical paths in design have been resolved, which improved performance from 10 MHz to 20 MHz, with the possibility of further improvement.

Then functions of the power sensor have been demonstrated by sampling two signals with an ADC and computing all desired values.

References

[1] IEEE. *IEEE Standard Definitions for the Measurement of Electric Power Quantities Under Sinusoidal, Nonsinusoidal, Balanced, or Unbalanced Conditions.* 2010, pp. 1–50. DOI: 10.1109/IEEESTD.2010.5439063.

[2] Svensson Stefan. "Power Measurement Techniques for Nonsinusoidal Conditions. The Significance of Harmonics for the Measurement of Power and other AC quantities". Doctoral thesis. Chalmers University of Technology, Göteborg, Sweden: CHALMERS UNIVERSITY OF TECHNOLOGY, 1999. URL: https://core.ac.uk/display/70557608?utm_source=pdf%5C&utm_medium=banner%5C&utm_campaign=pdf-decoration-v1 (visited on 04/09/2024).

[3] Allegro Microsystems Inc. *ACS37800. Datasheet.* Mar. 2022. URL: https://www.allegromicro.com/en/products/sense/current-sensor-ics/zero-to-fifty-amp-integrated-conductor-sensor-ics/acs37800.

[4] Shu-Chen Wang, Chi-Jui Wu, and Sheng-Wen Yang. "Applying FPGA-based chip to apparent power and power factor measurement considering nonsinusoidal and unbalanced conditions". In: *WSEAS Transactions on Circuits and Systems* 8 (July 2009), pp. 559–568.

[5] Aarenstrup Roger. *Managing Model-Based Design.* 1st ed. Apple Hill Drive 3, Natick, MA, United States: The MathWorks, Inc., 2015. ISBN: 978-1512036138.

[6] Babić Josip, Marijan Siniša, and Petrović Ivan. "Introducing Model-Based Techniques into Development of Real-Time Embedded Applications". In: *Automatika* 52.4 (Jan. 2017), pp. 329–338. ISSN: 0005-1144. DOI: 10.1080/00051144.2011.11828432. URL: https://www.tandfonline.com/doi/full/10.1080/00051144.2011.11828432 (visited on 04/08/2024).

[7] Synopsys Inc. *What is Model-Based Design?* 2024. URL: https://www.synopsys.com/glossary/what-is-model-based-design.html (visited on 04/08/2024).

[8] The MathWorks Inc. *HDL Coder.* URL: https://www.mathworks.com/products/hdl-coder.html (visited on 04/08/2024).

[9] Kyoji Marumoto and Hiroshi Nishide. *Improving the Efficiency of IC Development with Model-Based Design.* 2022. URL: https://www.mathworks.com/company/technical-articles/improving-the-efficiency-of-ic-development-with-model-based-design.html (visited on 04/08/2024).

[10] Analog Devices Inc. *Fixed-Point vs. Floating-Point Digital Signal Processing.* URL: https://www.analog.com/en/technical-articles/fixedpoint-vs-floatingpoint-dsp.html.

[11] IEEE. "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* pp.1-84 (July 2020). DOI: 10.1109/ieeestd.2019.8766229. URL: https://ieeexplore.ieee.org/document/8766229.

[12] Wallace Evan. *Float Toy.* URL: https://github.com/evanw/float-toy.

[13] The MathWorks Inc. *Benefits of Using Fixed-Point Hardware - MATLAB & Simulink.* www.mathworks.com. URL: https://www.mathworks.com/help/

simulink/ug/benefits-of-using-fixed-point-hardware.html (visited on 01/17/2024).

[14] Pyeatt Larry D. and Ughetta William. *ARM 64-Bit Assembly Language*. Newnes, 2020. ISBN: 978-0-12-819221-4. DOI: 10.1016/C2018-0-03846-3.

[15] Beheshti Babak D. “Fixed point performance of interpolation/extrapolation algorithms for resource constrained wireless sensors”. In: *2015 Long Island Systems, Applications and Technology* (2015), pp. 1–4. DOI: 10.1109/LISAT.2015.7160175. URL: http://ieeexplore.ieee.org/document/7160175/.

[16] Andraka Ray. “A survey of CORDIC algorithms for FPGA based computers”. In: *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays - FPGA ’98* (1998), pp. 191–200. DOI: 10.1145/275107.275139. URL: http://portal.acm.org/citation.cfm?doid=275107.275139.

[17] Volder Jack E. “The Birth of Cordic”. In: *The Journal of VLSI Signal Processing* 25.2 (2000), pp. 101–105. ISSN: 09225773. DOI: 10.1023/A:1008110704586. URL: http://link.springer.com/10.1023/A:1008110704586.

[18] uArt.cz. “Algoritmus CORDIC”. In: (2012). DOI: https://uart.cz/740/algoritmus-cordic/.

[19] Mopuri Suresh, Bhardwaj Swati, and Acharyya Amit. “Coordinate Rotation-Based Design Methodology for Square Root and Division Computation”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 66.7 (2019), pp. 1227–1231. ISSN: 1549-7747. DOI: 10.1109/TCSII.2018.2878599. URL: https://ieeexplore.ieee.org/document/8515067/.

[20] S. Walther. “A unified algorithm for elementary functions”. In: *Managing Requirements Knowledge, International Workshop on.* Vol. 1. Los Alamitos, CA, USA: IEEE Computer Society, May 1971, p. 379. DOI: 10.1109/AFIPS.1971.14. URL: https://doi.ieeecomputersociety.org/10.1109/AFIPS.1971.14.

[21] Libessart Erwan et al. “A scaling-less Newton-Raphson pipelined implementation for a fixed-point inverse square root operator”. In: *2017 15th IEEE International New Circuits and Systems Conference (NEWCAS)* (2017), pp. 157–160. DOI: 10.1109/NEWCAS.2017.8010129. URL: http://ieeexplore.ieee.org/document/8010129/ (visited on 01/18/2024).

[22] Hertz Erik et al. “Algorithms for implementing roots, inverse and inverse roots in hardware”. In: 2016. URL: https://api.semanticscholar.org/CorpusID:2183317.

[23] Shen-Fu Hsiao, Chia-Sheng Wen, and Ming-Yu Tsai. “Low-cost design of reciprocal function units using shared multipliers and adders for polynomial approximation and Newton Raphson interpolation”. In: *2010 International Symposium on Next Generation Electronics* (2010), pp. 40–43. DOI: 10.1109/ISNE.2010.5669204. URL: http://ieeexplore.ieee.org/document/5669204/ (visited on 01/18/2024).

[24] Trefethen Lloyd N. *Approximation Theory and Approximation Practice, Extended Edition.* Philadelphia, PA: Society for Industrial and Applied Mathematics, 2019. DOI: 10.1137/1.9781611975949. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611975949. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611975949.

[25] Navara Mirko and Němeček Aleš. *Numerické metody.* Praha: Vydavatelství ČVUT, 2003. ISBN: 80-010-2689-2.

[26] Kitamoto Takuya. "On the Computation of the Determinant of a Generalized Vandermonde Matrix". In: *Computer Algebra in Scientific Computing* (2014), pp. 242–255. DOI: 10.1007/978-3-319-10515-4_18. URL: http://link.springer.com/10.1007/978-3-319-10515-4_18 (visited on 01/18/2024).

[27] Meyer-Bäse Uwe. *Digital signal processing with field programmable gate arrays.* 3rd ed. Berlin: Springer, 2007. ISBN: 978-3-540-72612-8.

[28] Ewart Timothée et al. "Polynomial Evaluation on Superscalar Architecture, Applied to the Elementary Function e x". In: *ACM Transactions on Mathematical Software* 46.3 (2020), pp. 1–22. ISSN: 0098-3500. DOI: 10.1145/3408893. URL: https://dl.acm.org/doi/10.1145/3408893 (visited on 01/18/2024).

[29] Louise H. Crockett et al. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc.* Glasgow, GBR: Strathclyde Academic Media, 2014. ISBN: 099297870X.

[30] Xilinx Inc. *Zynq-7000 SoC Data Sheet: Overview (DS190). Datasheet.* July 2018.

[31] Xilinx Inc. *Series FPGAs and Zynq-7000 SoC XADC Dual 12-Bit 1 MSPS Analog-to-Digital Converter User Guide (UG480). User Guide.* June 2022.

[32] Avnet Inc. *ZedBoard Zynq™Evaluation and Development Hardware User's Guide. User Guide.* Aug. 2012.

[33] Schmid Christian and Peterer Roland. "Evelopment Tools for the Xilinx Zynq-7000 SoC. Design of a Development Framework and System Interaction Method for Embedded Systems on the Zynq-7000 SoC". Master thesis. Trondheim: NTNU: Norwegian University of Science and Technology, 2016. URL: http://hdl.handle.net/11250/2404384.

[34] The Mathworks Inc. *Getting Started with Targeting Xilinx Zynq Platform.* URL: https://www.mathworks.com/help/hdlcoder/ug/getting-started-with-hardware-software-codesign-workflow-for-xilinx-zynq-platform.html%5C#d122e108434. (visited on 04/22/2024).

[35] The Mathworks Inc. *Guided Hardware Setup.* URL: https://www.mathworks.com/help/hdlcoder/xilinxzynq7000/ug/guided-sd-card-setup.html (visited on 04/22/2024).

[36] The Mathworks Inc. *Zynq.* URL: https://www.mathworks.com/help/ecoder/xilinxzynq7000ec/ref/zynq.html (visited on 04/22/2024).

[37] Xilinx Inc. *Vivado Design Suite User Guide: Design Flows Overview (UG892). User Guide.* Oct. 2022. URL: https://docs.amd.com/r/en-US/ug892-vivado-design-flows-overview.

[38] Xilinx Inc. *Vivado Design Suite User Guide: Design Analysis and Closure Techniques (UG906). User Guide.* May 2021. URL: https://docs.amd.com/v/u/2019.1-English/ug906-vivado-design-analysis.

[39] The Mathworks Inc. *Define Custom Board and Reference Design for Zynq Workflow.* URL: https://www.mathworks.com/help/hdlcoder/ug/define-and-register-

custom-board-and-reference-design-for-zynq-workflow.html (visited on 04/26/2024).

[40] The Mathworks Inc. *Distributed Pipelining: Speed Optimization.* URL: https://www.mathworks.com/help/hdlcoder/ug/distributed-pipelining-speed-optimization.html (visited on 04/27/2024).

[41] STMicroelectronics. *STM32F303xD STM32F303xE. Datasheet.* Oct. 2016. URL: https://www.st.com/resource/en/datasheet/stm32f303re.pdf.

[42] STMicroelectronics. *STM32CubeIDE. Integrated Development Environment for STM32.* URL: https://www.st.com/en/development-tools/stm32cubeide.html (visited on 05/17/2024).

[43] Digilent Inc. *ZED. Zedboard Schematic.* June 2020. URL: https://digilent.com/reference/_media/reference/programmable-logic/zedboard/zedboard-schematic-rev-e1-public.pdf.

Appendices

A Simulated values - Average and Apparent Power

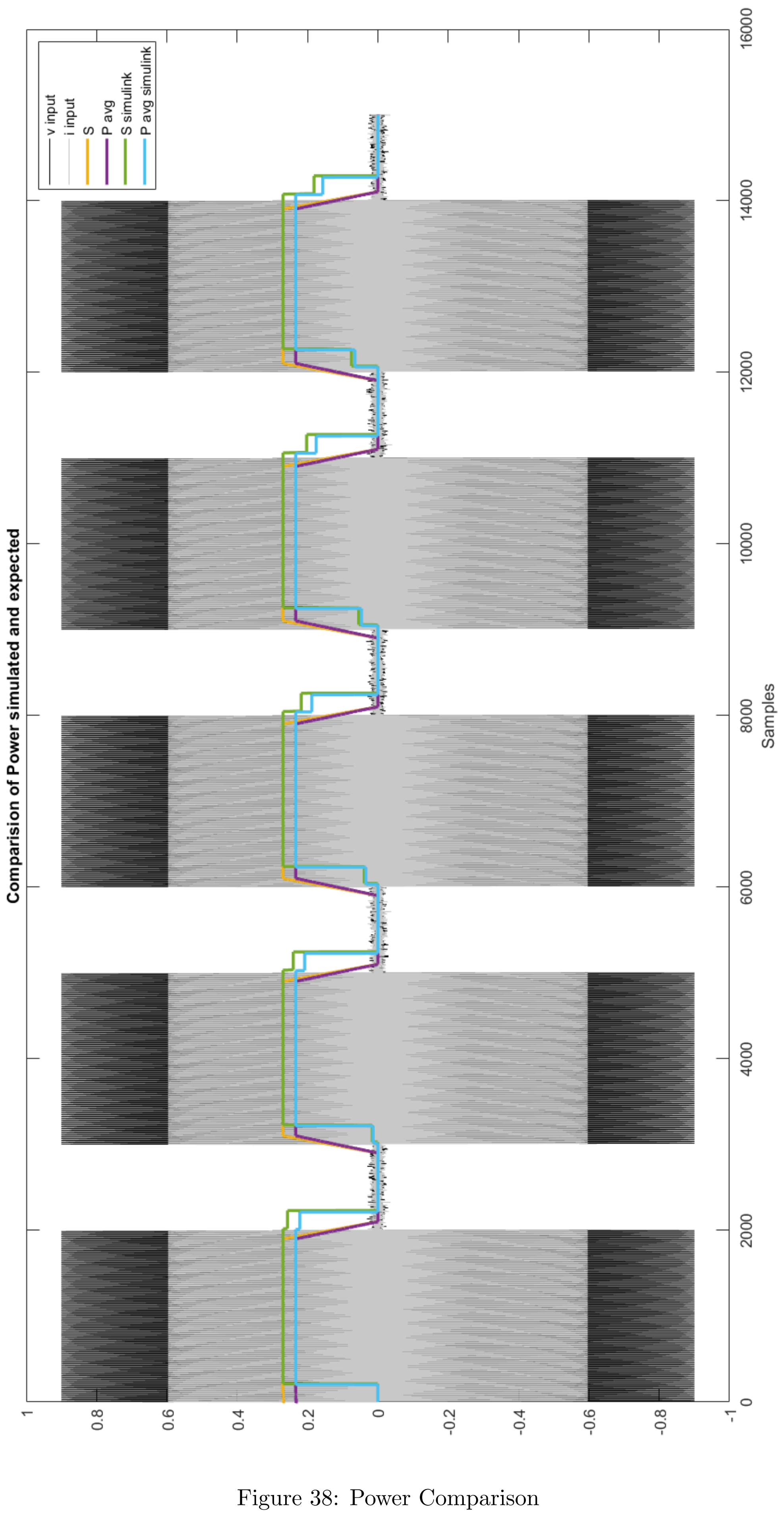


Figure 38: Power Comparison

B Simulated RMS values

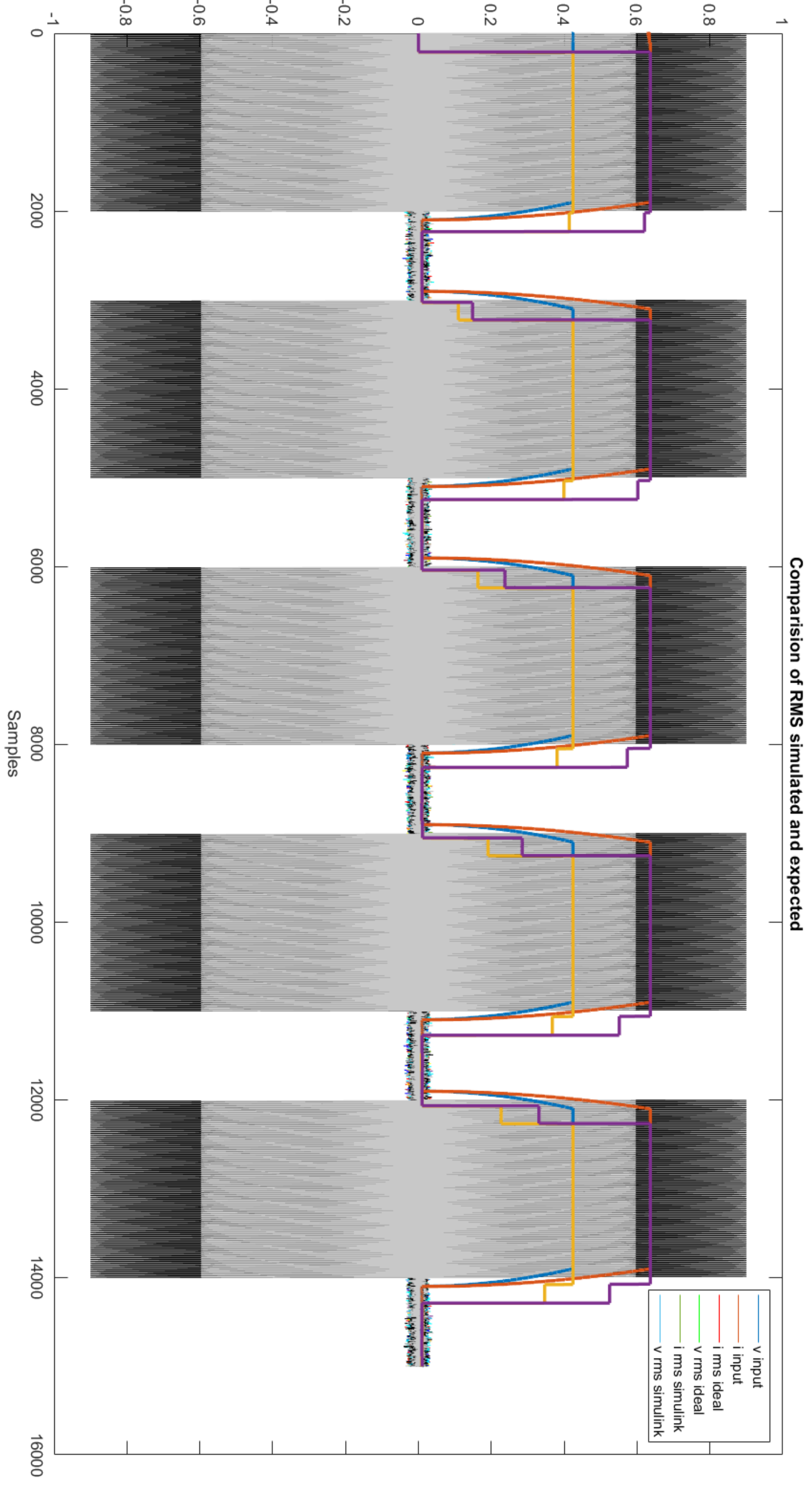


Figure 39: RMS Comparison

C Measured results with window change

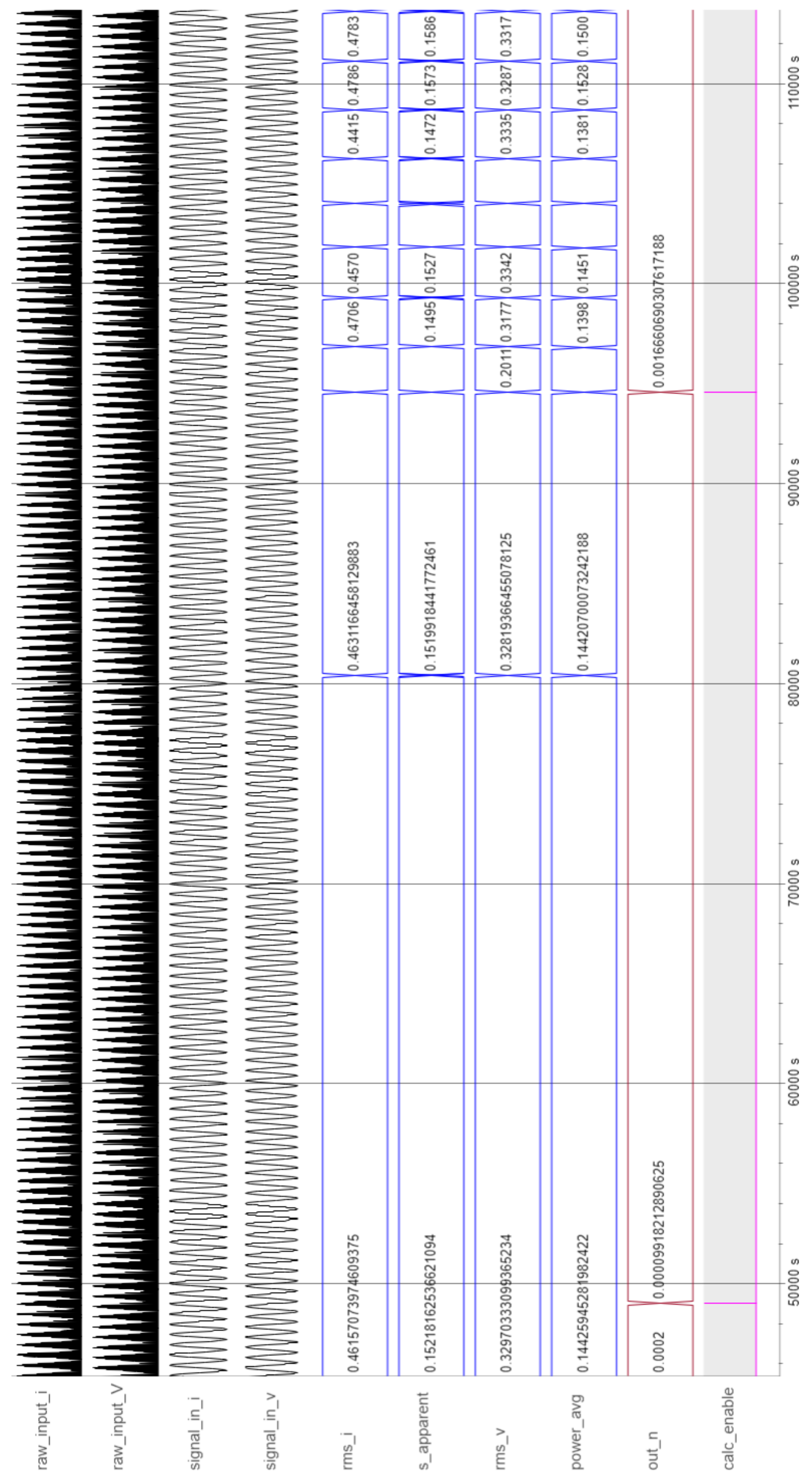


Figure 40: Measured result with window change

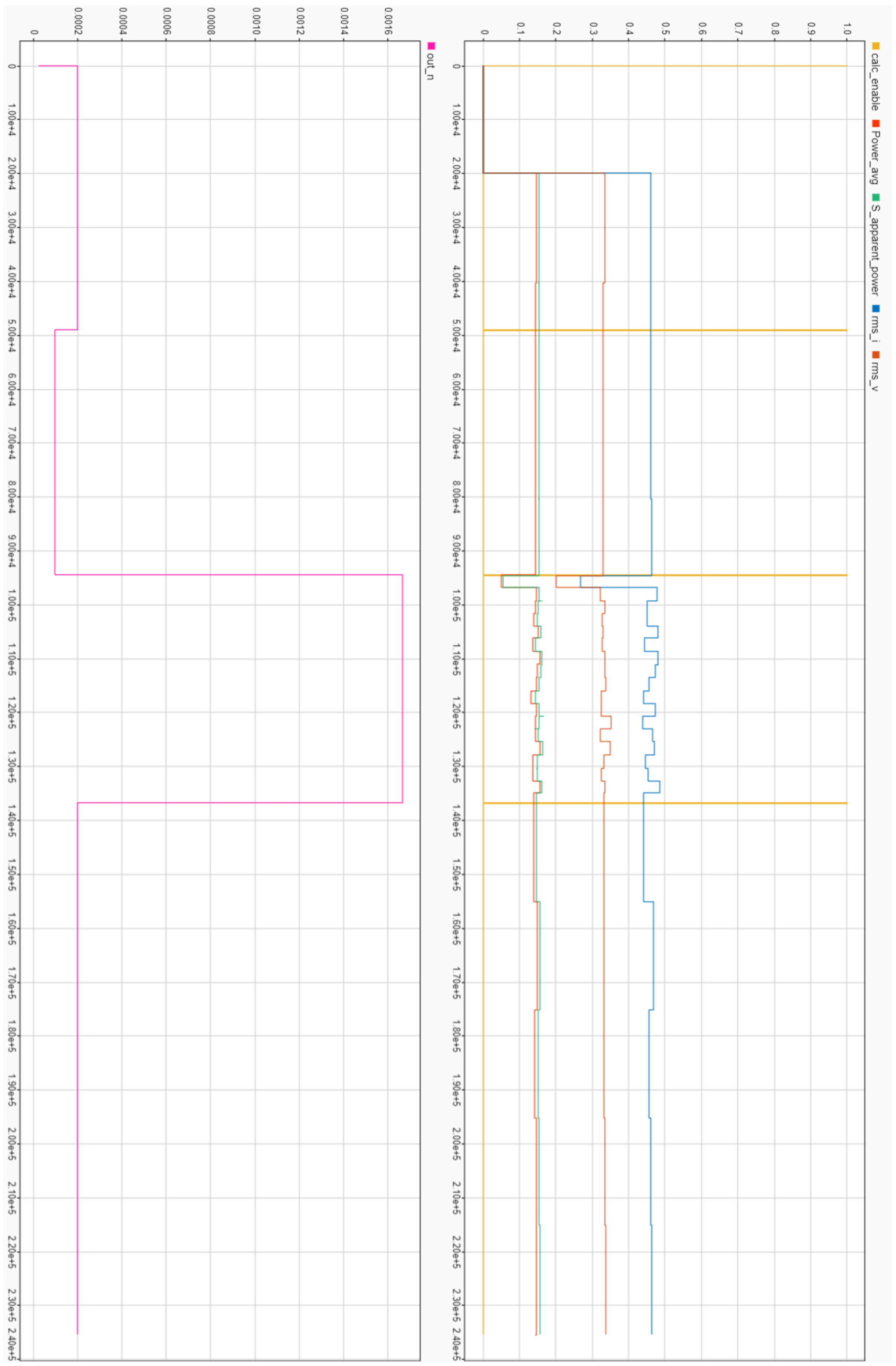


Figure 41: Measured result with window change

D FIL Simulation Compare subsystem

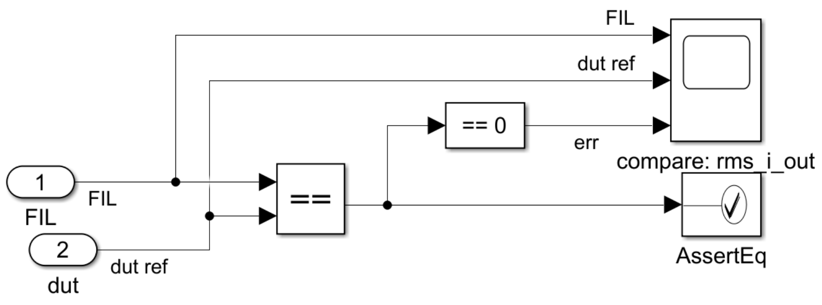
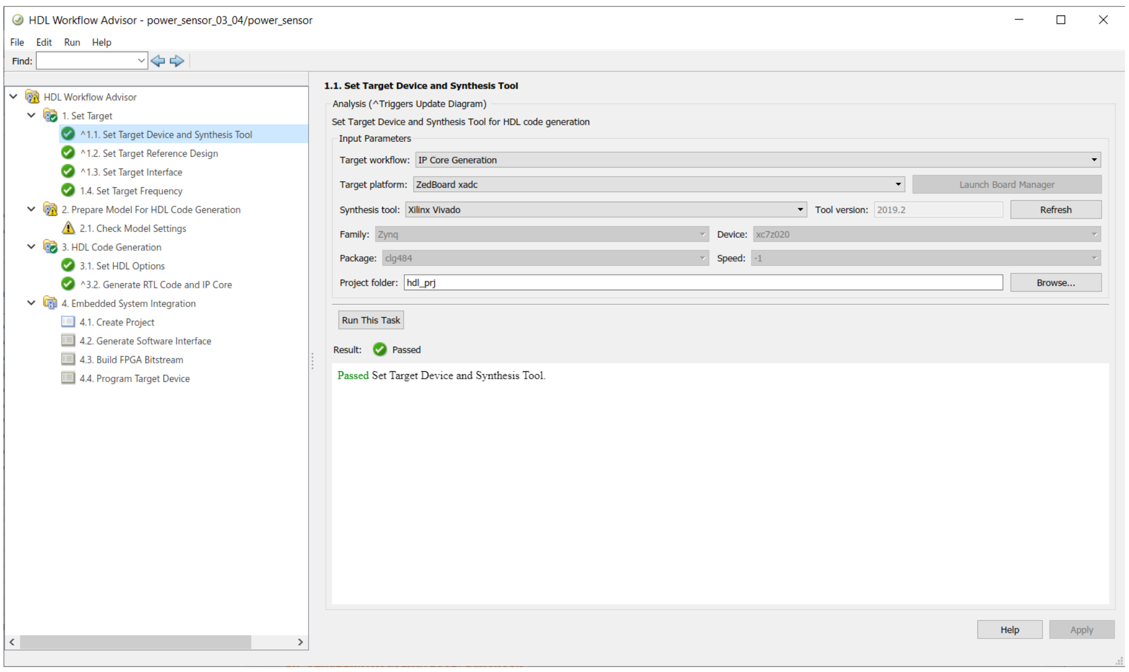


Figure 42: FIL Simulation Compare subsystem

E HDL Coder Workflow Advisor example walktrough


HDL Workflow Advisor - power_sensor_03_04/power_sensor

File Edit Run Help

Find:

- HDL Workflow Advisor
  - 1. Set Target
    - ^1.1. Set Target Device and Synthesis Tool
    - ^1.2. Set Target Reference Design
    - ^1.3. Set Target Interface
    - 1.4. Set Target Frequency
  - 2. Prepare Model For HDL Code Generation
    - 2.1. Check Model Settings
  - 3. HDL Code Generation
    - 3.1. Set HDL Options
    - ^3.2. Generate RTL Code and IP Core
  - 4. Embedded System Integration
    - 4.1. Create Project
    - 4.2. Generate Software Interface
    - 4.3. Build FPGA Bitstream
    - 4.4. Program Target Device

1.1. Set Target Device and Synthesis Tool

Analysis (^Triggers Update Diagram)

Set Target Device and Synthesis Tool for HDL code generation

Input Parameters

Target workflow: IP Core Generation

Target platform: ZedBoard xadc — Launch Board Manager

Synthesis tool: Xilinx Vivado — Tool version: 2019.2 — Refresh

Family: Zynq — Device: xc7z020

Package: clg484 — Speed: -1

Project folder: hdl_prj — Browse...

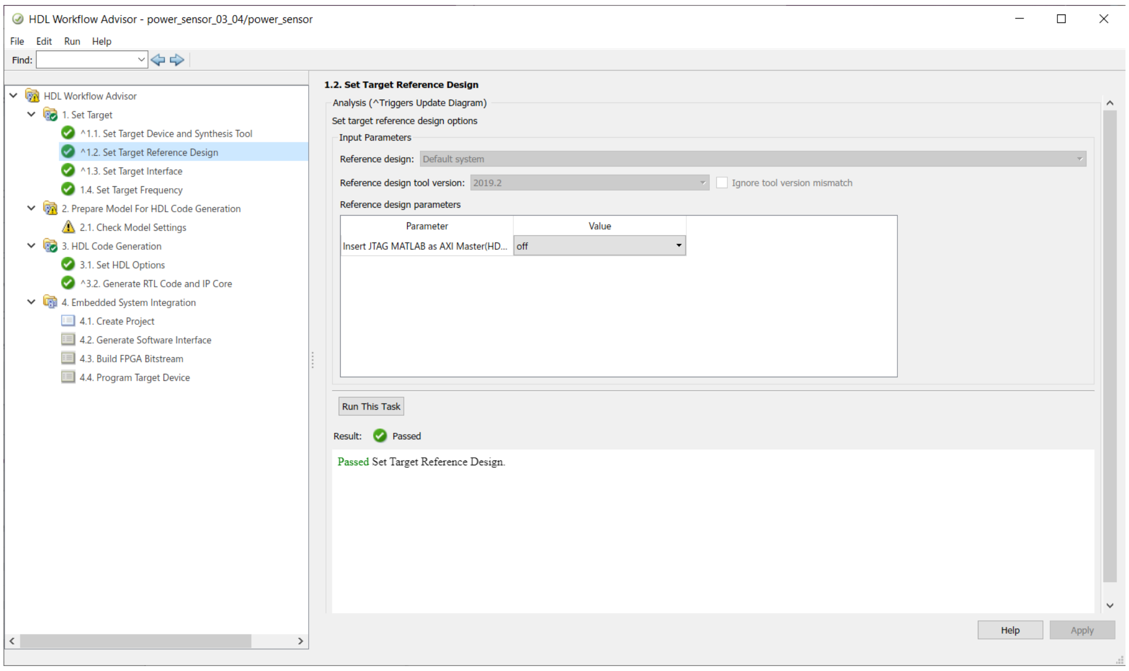
Run This Task

Result: Passed

Passed Set Target Device and Synthesis Tool.

Help Apply

Figure 43: HDL Coder Workflow Advisor step 1.1


HDL Workflow Advisor - power_sensor_03_04/power_sensor
File Edit Run Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthesis Tool
^1.2. Set Target Reference Design
^1.3. Set Target Interface
1.4. Set Target Frequency
2. Prepare Model For HDL Code Generation
2.1. Check Model Settings
3. HDL Code Generation
3.1. Set HDL Options
^3.2. Generate RTL Code and IP Core
4. Embedded System Integration
4.1. Create Project
4.2. Generate Software Interface
4.3. Build FPGA Bitstream
4.4. Program Target Device
Run This Task
Help
Apply

1.2. Set Target Reference Design

Analysis (^Triggers Update Diagram)

Set target reference design options

Input Parameters

Reference design: Default system

Reference design tool version: 2019.2 ☐ Ignore tool version mismatch

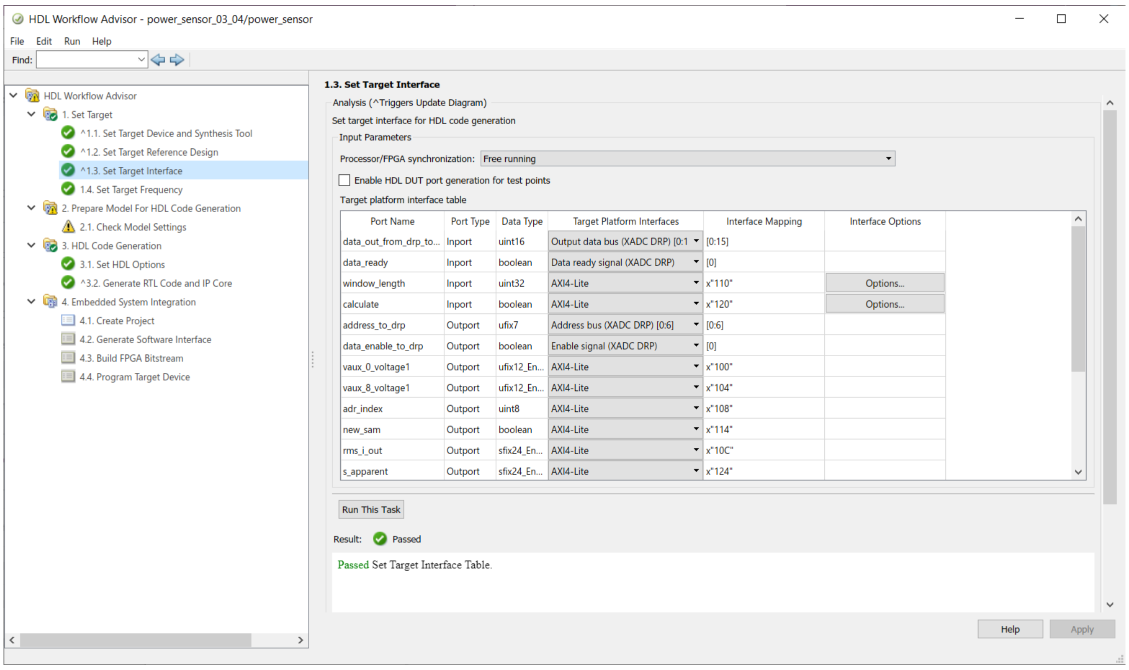
Reference design parameters

Parameter	Value
Insert JTAG MATLAB as AXI Master(HD...	off

Result: ✓ Passed

Passed Set Target Reference Design.

Figure 44: HDL Coder Workflow Advisor step 1.2


HDL Workflow Advisor - power_sensor_03_04/power_sensor
File Edit Run Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthesis Tool
^1.2. Set Target Reference Design
^1.3. Set Target Interface
1.4. Set Target Frequency
2. Prepare Model For HDL Code Generation
2.1. Check Model Settings
3. HDL Code Generation
3.1. Set HDL Options
^3.2. Generate RTL Code and IP Core
4. Embedded System Integration
4.1. Create Project
4.2. Generate Software Interface
4.3. Build FPGA Bitstream
4.4. Program Target Device
Run This Task
Help
Apply

1.3. Set Target Interface

Analysis (^Triggers Update Diagram)

Set target interface for HDL code generation

Input Parameters

Processor/FPGA synchronization: Free running

☐ Enable HDL DUT port generation for test points

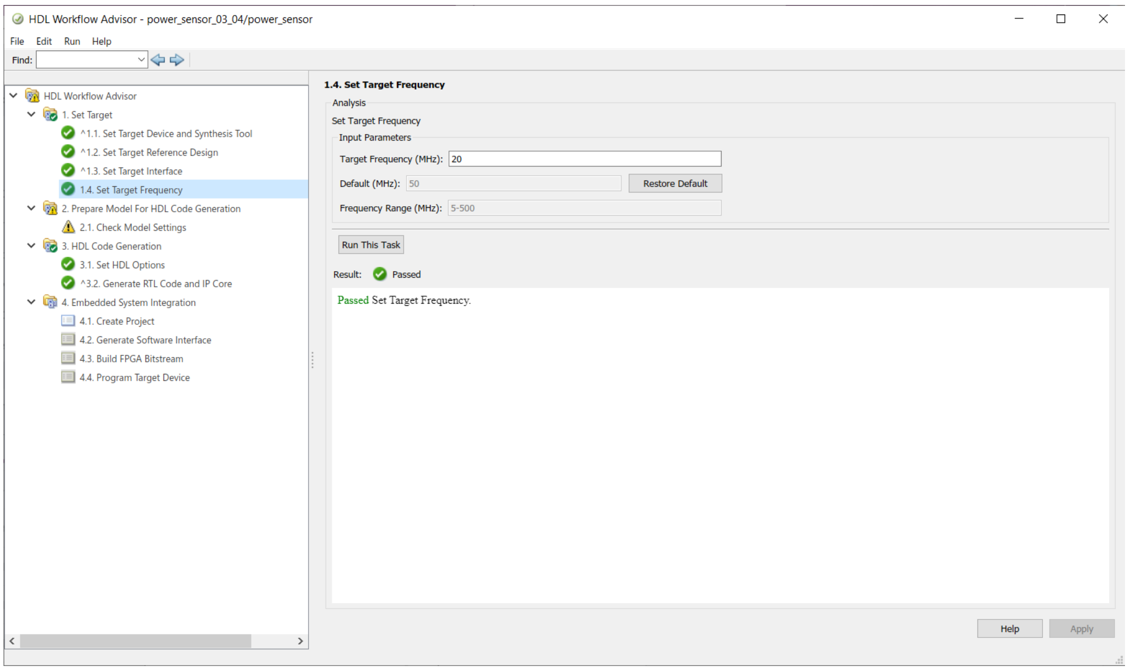
Target platform interface table

Port Name	Port Type	Data Type	Target Platform Interfaces	Interface Mapping	Interface Options
data_out_from_drp_to...	Inport	uint16	Output data bus (XADC DRP) [0:1	[0:15]	
data_ready	Inport	boolean	Data ready signal (XADC DRP)	[0]	
window_length	Inport	uint32	AXI4-Lite	x"110"	Options...
calculate	Inport	boolean	AXI4-Lite	x"120"	Options...
address_to_drp	Outport	ufix7	Address bus (XADC DRP) [0:6]	[0:6]	
data_enable_to_drp	Outport	boolean	Enable signal (XADC DRP)	[0]	
vaux_0_voltage1	Outport	ufix12_En...	AXI4-Lite	x"100"	
vaux_8_voltage1	Outport	ufix12_En...	AXI4-Lite	x"104"	
adr_index	Outport	uint8	AXI4-Lite	x"108"	
new_sam	Outport	boolean	AXI4-Lite	x"114"	
rms_i_out	Outport	sfix24_En...	AXI4-Lite	x"10C"	
s_apparent	Outport	sfix24_En...	AXI4-Lite	x"124"	

Result: ✓ Passed

Passed Set Target Interface Table.

Figure 45: HDL Coder Workflow Advisor step 1.3


HDL Workflow Advisor - power_sensor_03_04/power_sensor
File Edit Run Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthesis Tool
^1.2. Set Target Reference Design
^1.3. Set Target Interface
1.4. Set Target Frequency
2. Prepare Model For HDL Code Generation
2.1. Check Model Settings
3. HDL Code Generation
3.1. Set HDL Options
^3.2. Generate RTL Code and IP Core
4. Embedded System Integration
4.1. Create Project
4.2. Generate Software Interface
4.3. Build FPGA Bitstream
4.4. Program Target Device
Restore Default
Run This Task
Help
Apply

1.4. Set Target Frequency

Analysis

Set Target Frequency

Input Parameters

Target Frequency (MHz): 20

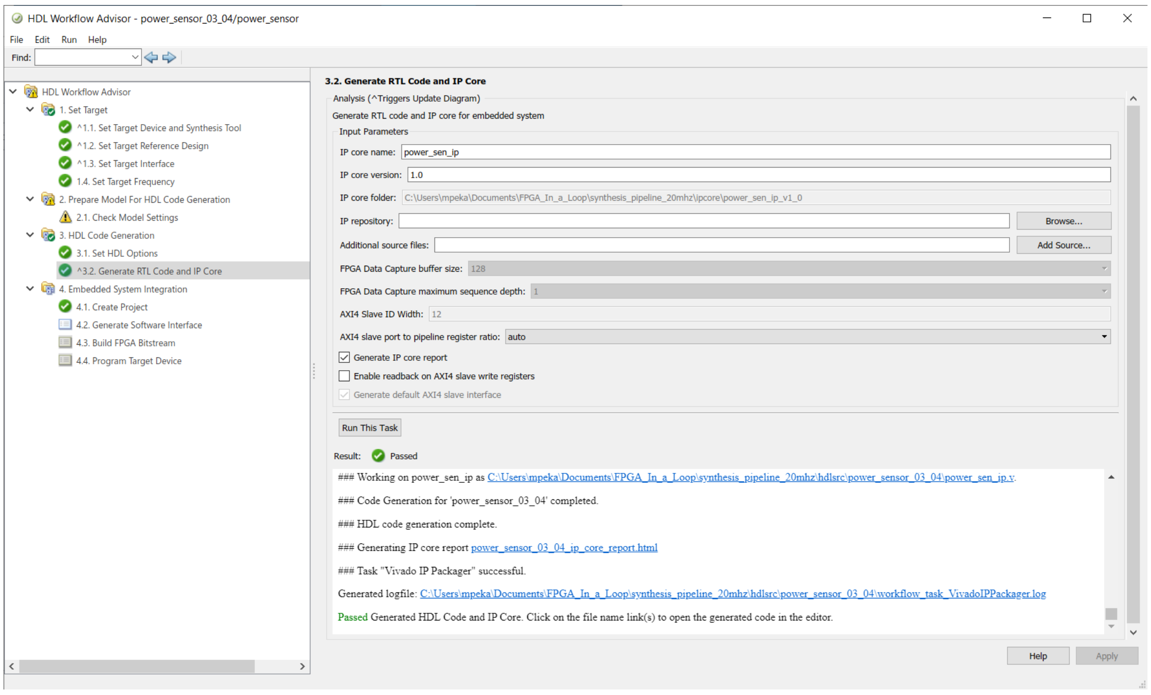
Default (MHz): 50

Frequency Range (MHz): 5-500

Result: Passed

Passed Set Target Frequency.

Figure 46: HDL Coder Workflow Advisor step 1.4


HDL Workflow Advisor - power_sensor_03_04/power_sensor
File Edit Run Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthesis Tool
^1.2. Set Target Reference Design
^1.3. Set Target Interface
1.4. Set Target Frequency
2. Prepare Model For HDL Code Generation
2.1. Check Model Settings
3. HDL Code Generation
3.1. Set HDL Options
^3.2. Generate RTL Code and IP Core
4. Embedded System Integration
4.1. Create Project
4.2. Generate Software Interface
4.3. Build FPGA Bitstream
4.4. Program Target Device
Browse...
Add Source...
Run This Task
Help
Apply

3.2. Generate RTL Code and IP Core

Analysis (^Triggers Update Diagram)

Generate RTL code and IP core for embedded system

Input Parameters

IP core name: power_sen_ip

IP core version: 1.0

IP core folder: C:\Users\mpeka\Documents\FPGA_In_a_Loop\synthesis_pipeline_20mhz\ipcore\power_sen_ip_v1_0

IP repository:

Additional source files:

FPGA Data Capture buffer size: 128

FPGA Data Capture maximum sequence depth: 1

AXI4 Slave ID Width: 12

AXI4 slave port to pipeline register ratio: auto

[✓] Generate IP core report

[ ] Enable readback on AXI4 slave write registers

[✓] Generate default AXI4 slave interface

Result: Passed

```
### Working on power_sen_ip as C:\Users\mpeka\Documents\FPGA_In_a_Loop\synthesis_pipeline_20mhz\hdlsrc\power_sensor_03_04\power_sen_ip.v.

### Code Generation for 'power_sensor_03_04' completed.

### HDL code generation complete.

### Generating IP core report power_sensor_03_04_ip_core_report.html

### Task "Vivado IP Packager" successful.

Generated logfile: C:\Users\mpeka\Documents\FPGA_In_a_Loop\synthesis_pipeline_20mhz\hdlsrc\power_sensor_03_04\workflow_task_VivadoIPPackager.log

Passed Generated HDL Code and IP Core. Click on the file name link(s) to open the generated code in the editor.
```

Figure 47: HDL Coder Workflow Advisor step 3.2

F script *hdlcoder_board_customization.m* for registering custom reference design

```matlab
function r = hdlcoder_board_customization
% Board plugin registration file
% 1. Any registration file with this name on MATLAB path will be picked up
% 2. Registration file returns a cell array pointing to the location of
%    the board plug-ins
% 3. Board plugin must be a package folder accessible from MATLAB path,
%    and contains a board definition file

%   Copyright 2012-2018 The MathWorks, Inc.

r = { ...
'ZCU102.plugin_board', ...
'ZynqZC702.plugin_board', ...
'ZynqZC706.plugin_board', ...
'ZedBoard.plugin_board', ...
'my_zedboard.plugin_board', ...
'my_zedboard_temp.plugin_board', ....
'my_zedboard2.plugin_board', ...
'XADCDemo.plugin_board', ...
'ZedBoard_xadc.plugin_board', ...
'ZedBoard_xadc_stream.plugin_board', ...
'ZedBoardxadcstream.plugin_board', ...
};

end
% LocalWords:  Zynq ZC ZCU
```

G script *hdlcoder_ref_design_customization.m* for registering custom reference design

```matlab
function [rd, boardName] = hdlcoder_ref_design_customization
% Reference design plugin registration file
% 1. The registration file with this name inside of a board plugin folder
%    will be picked up
% 2. Any registration file with this name on MATLAB path will also be picked up
% 3. The registration file returns a cell array pointing to the location of
%    the reference design plugins
% 4. The registration file also returns its associated board name
% 5. Reference design plugin must be a package folder accessible from
%    MATLAB path, and contains a reference design definition file

%   Copyright 2012-2018 The MathWorks, Inc.

rd = {...
    'ZedBoard_xadc.vivado_base_2018_2.plugin_rd', ...
    'ZedBoard_xadc.vivado_stream_2018_2.plugin_rd', ...
    };

boardName = 'ZedBoard xadc';

end
% LocalWords:  Zynq ZC edk vivado
```

H script *plugin_baord.m* for registering custom reference design

```
function hB = plugin_board()
% Board definition

%   Copyright 2012-2014 The MathWorks, Inc.

% Construct board object
hB = hdlcoder.Board;

hB.BoardName     = 'ZedBoard xadc';

% FPGA device information
hB.FPGAVendor    = 'Xilinx';
hB.FPGAFamily    = 'Zynq';
hB.FPGADevice    = 'xc7z020';
hB.FPGAPackage   = 'clg484';
hB.FPGASpeed     = '-1';

% Tool information
hB.SupportedTool = {'Xilinx Vivado'};

% FPGA JTAG chain position
hB.JTAGChainPosition = 2;

%% Add interfaces
% Standard "External Port" interface
hB.addExternalPortInterface( ...
    'IOPadConstraint', {'IOSTANDARD = LVCMOS18'});

% Custom board external I/O interface
hB.addExternalIOInterface( ...
    'InterfaceID',    'LEDs General Purpose', ...
    'InterfaceType',  'OUT', ...
    'PortName',       'GPLEDs', ...
    'PortWidth',      8, ...
    'FPGAPin',        {'T22', 'T21', 'U22', 'U21', 'V22', '
      W22', 'U19', 'U14'}, ...
    'IOPadConstraint', {'IOSTANDARD = LVCMOS33'});

hB.addExternalIOInterface( ...
    'InterfaceID',    'DIP Switches', ...
    'InterfaceType',  'IN', ...
    'PortName',       'DIPSwitches', ...
    'PortWidth',      8, ...
    'FPGAPin',        {'F22', 'G22', 'H22', 'F21', 'H19', '
      H18', 'H17', 'M15'}, ...
    'IOPadConstraint', {'IOSTANDARD = LVCMOS25'});
```

```

hB.addExternalIOInterface( ...
    'InterfaceID',    'Push Buttons L-R-U-D-S', ...
    'InterfaceType',  'IN', ...
    'PortName',       'PushButtons', ...
    'PortWidth',      5, ...
    'FPGAPin',        {'N15', 'R18', 'T18', 'R16', 'P16'}, ...
    'IOPadConstraint', {'IOSTANDARD = LVCMOS25'});

hB.addExternalIOInterface( ...
    'InterfaceID',    'Pmod Connector JA1', ...
    'InterfaceType',  'INOUT', ...
    'PortName',       'PmodJA1', ...
    'PortWidth',      8, ...
    'FPGAPin',        {'Y11', 'AA11', 'Y10', 'AA9', 'AB11', 'AB10', 'AB9', 'AA8'}, ...
    'IOPadConstraint', {'IOSTANDARD = LVCMOS33'});

hB.addExternalIOInterface( ...
    'InterfaceID',    'Pmod Connector JB1', ...
    'InterfaceType',  'INOUT', ...
    'PortName',       'PmodJB1', ...
    'PortWidth',      8, ...
    'FPGAPin',        {'W12', 'W11', 'V10', 'W8', 'V12', 'W10', 'V9', 'V8'}, ...
    'IOPadConstraint', {'IOSTANDARD = LVCMOS33'});

hB.addExternalIOInterface( ...
    'InterfaceID',    'Pmod Connector JC1', ...
    'InterfaceType',  'INOUT', ...
    'PortName',       'PmodJC1', ...
    'PortWidth',      8, ...
    'FPGAPin',        {'AB7', 'AB6', 'Y4', 'AA4', 'R6', 'T6', 'T4', 'U4'}, ...
    'IOPadConstraint', {'IOSTANDARD = LVCMOS33'});

hB.addExternalIOInterface( ...
    'InterfaceID',    'Pmod Connector JD1', ...
    'InterfaceType',  'INOUT', ...
    'PortName',       'PmodJD1', ...
    'PortWidth',      8, ...
    'FPGAPin',        {'V7', 'W7', 'V5', 'V4', 'W6', 'W5', 'U6', 'U5'}, ...
    'IOPadConstraint', {'IOSTANDARD = LVCMOS33'});
```

I script *plugin_rd* for registering custom reference design

```
function hRD = plugin_rd()
% Reference design definition
```

```matlab

%   Copyright 2014-2020 The MathWorks, Inc.

% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx
   Vivado');

hRD.ReferenceDesignName = 'Default system';
hRD.BoardName = 'ZedBoard xadc';

% Tool information
hRD.SupportedToolVersion = {'2018.2','2018.3','2019.1','
   2019.2','2020.1'};

%% Add custom design files
% add custom Vivado design
hRD.addCustomVivadoDesign( ...
    'CustomBlockDesignTcl', 'system_top.tcl', ...
    'VivadoBoardPart',      'em.avnet.com:zed:part0:1.0');

%% Add custom files and constraint files

%% Add interfaces
% add clock interface
hRD.addClockInterface( ...
    'ClockConnection',     'core_clkwiz/clk_out1', ...
    'ResetConnection',     'sys_core_rstgen/
       peripheral_aresetn',...
    'DefaultFrequencyMHz', 50,...
    'MinFrequencyMHz',     5,...
    'MaxFrequencyMHz',     500,...
    'ClockModuleInstance', 'core_clkwiz',...
    'ClockNumber',         1);

% add AXI4 and AXI4-Lite slave interfaces
hRD.addAXI4SlaveInterface( ...
    'InterfaceConnection', 'axi_cpu_interconnect/M00_AXI', ...
    'BaseAddress',         '0x400D0000', ...
    'MasterAddressSpace',  'sys_cpu/Data');

hRD.DeviceTreeName = 'devicetree_axilite_iio.dtb';

% add interface to additional IP in the reference design
hRD.addInternalIOInterface( ...
    'InterfaceID',      'Output data bus (XADC DRP)', ...
    'InterfaceType',    'IN', ...
    'PortName',         'XADC_DO_OUT', ...
    'PortWidth',        16, ...
    'InterfaceConnection',  'xadc_wiz_0/do_out');
```

```
hRD.addInternalIOInterface( ...
    'InterfaceID',      'Data ready signal (XADC DRP)', ...
    'InterfaceType',    'IN', ...
    'PortName',         'XADC_DRDY_OUT', ...
    'PortWidth',        1, ...
    'InterfaceConnection',  'xadc_wiz_0/drdy_out');
hRD.addInternalIOInterface( ...
    'InterfaceID',      'Address bus (XADC DRP)', ...
    'InterfaceType',    'OUT', ...
    'PortName',         'XADC_DADDR_IN', ...
    'PortWidth',        7, ...
    'InterfaceConnection',  'xadc_wiz_0/daddr_in');
hRD.addInternalIOInterface( ...
    'InterfaceID',      'Enable signal (XADC DRP)', ...
    'InterfaceType',    'OUT', ...
    'PortName',         'XADC_DEN_IN', ...
    'PortWidth',        1, ...
    'InterfaceConnection',  'xadc_wiz_0/den_in');


% LocalWords:  Zynq ZC vlnv xilinx zynq zc AXI axi Addr wiz
   aresetn IPCORE
% LocalWords:  avnet devicetree axilite dtb Vivado
```

J Top level of generated model for Embedded Coder and measured data transfer

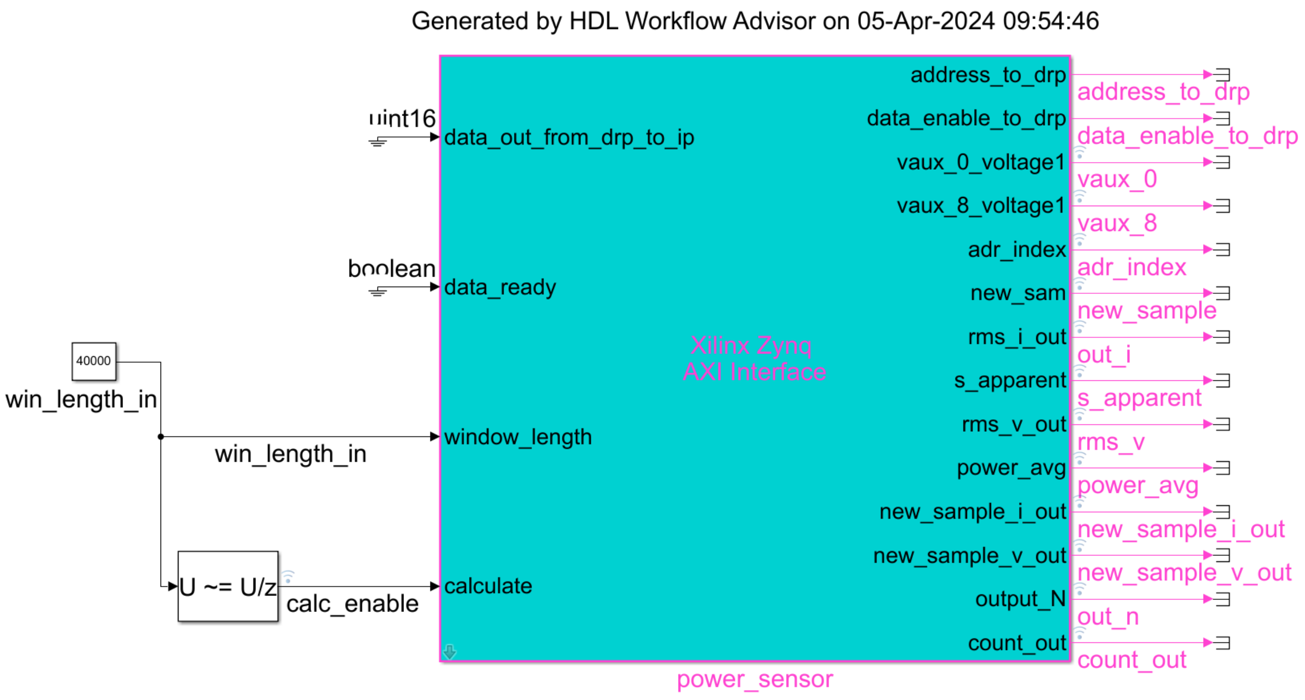


Figure 48: Top level of generated model for Embedded Coder and measured data transfer.

K Submodule *AXI4SlaveRead* of generated model

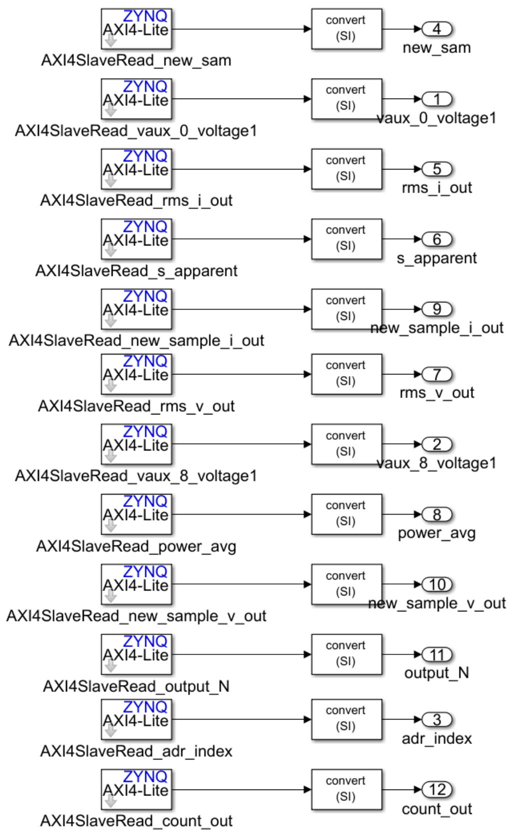


Figure 49: Submodule *AXI4SlaveRead* of generated model