Expertise
and insight
for the future

PHUOC TAI VO

Digital Design and Verification of Adaptive Noise Cancellation Module

Metropolia University of Applied Sciences

Bachelor of Engineering

Degree Programme in Electronics Engineering

Bachelor's Thesis

10 November 2020

Author	PHUOC TAI VO
Title	Digital Design and Verification of Adaptive Noise Cancellation
Number of Pages	61 pages
Date	10 November 2020
Degree	Bachelor of Engineering
Degree Programme	Electronics Engineering
Specialisation option	Electronics
Instructor(s)	Janne Mäntykoski, Senior Lecturer

Currently, at Metropolia University Applied of Sciences, digital design and implementation based on DSP studies have mainly focused on how to write efficient codes according to common coding styles rule. However, for its verification process before physical design, verification with an HDL testbench is the only method which is instructed in university. Advanced and high-level verification methods, especially UVM, are still not applied for teaching and studies. Apart from that, when it comes to DSP implementation, adaptive filtering approaches were selected to solve the interesting problem of how to track the statistical variations of signal and eliminate random noise over time in a nonstationary environment with massive datasets and unstructured data types for complex hardware implementation. Therefore, the Thesis work aimed to design and implement an adaptive FIR filtering noise cancellation module based on LMS algorithm and verify its functional specifications.

In the Thesis work, an adaptive noise cancellation module was developed to be well-prepared for hardware emulation. Module design and verification studied MathWorks workflow step by step. Matlab reference code was designed and optimized on algorithm level. Simulink model which contains the fixed-point implementation was simulated to be compared with Matlab algorithm for periodic discrete-time environment before synthesizing into VHDL. The Register Transfer Level Design was verified in cosimulation with Simulink model. Besides, UVM based verification was applied to use System Verilog as verification language to evaluate the fulfillment of functional requirements and guarantee design accuracy with bit-cycle level.

The Thesis presents the design module, which serves the purpose of noise cancellation based on adaptive LMS algorithm. The theory and mathematical model of the design module are explained. RTL reusability was achieved with the use of modular coding style to create instantiated functions. The implementation work was done adhering to Mathworks digital design flow. Additionally, no errors or bugs were found from the design module under verifications. The module passed all required functionality tests. Hardware deployment and the related challenges, such as IP core generation to be integrated with a register abstraction layer, could be innovated for future development of this Thesis project, since the design module was already modified to be modular and reusable to form a scalable architecture with generic parameter specifications.

Keywords	Adaptive filter, UVM, finite impulse response, digital design, RTL

Contents

List of Abbreviations

ASIC	Application Specific Integrated Circuit
AXI	Advanced eXtensible Interface
DPI	Direct Programming Interface
DSP	Digital Signal Processing
DUT	Design Under Test
DUV	Design Under Verification
EDA	Electronic Design Automation
FEC	Focused Expression Coverage
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
IEEE	Institute of Electrical and Electronics Engineers
IIR	Infinite Impulse Response
IP	Intellectual Property
LMS	Least Mean Squares
RLS	Recursive Least Squares
RTL	Register Transfer Level
SoC	System On Chip

SV	System Verilog
VHDL	Very High Speed Integrated Circuit Hardware Design Language
UVM	Universal Verification Methodology

1 Introduction

For the decades, the tremendous development in digital signal processing area which is in relation to growing technologies allows digital design and hardware implementation of signal processing to practically implement mathematical computations and complex algorithms. If the signal statistics are available to define data properties, fixed algorithms could be appropriately selected to be processed easily. Despite that, to track the statistical variations of signal over time in a nonstationary environment, its algorithms to be determined in advance are no longer efficient. The solution for this challenge is to use adaptive filter that automatically adapts to input characteristics change and noise cancellation by iterative estimation of configured parameters. Therefore, from this motivation, the Thesis work was to implemented to present the overall process of how to set up the adaptive noise cancellation module step-by-step from fixed-point conversion, reference modeling, RTL design with HDL testbench integration and UVM-based verification.

The goal of the Thesis work was to design, implement and verify an adaptive FIR filtering noise cancellation module coded in hardware description language according to the predefined design specifications and reference algorithms written by Matlab. The design had to be configurable with modular coding style that it can be reused in future university studies or innovation projects. Moreover, the design module is bit-exactly verified in UVM test bench against the implemented fixed-point Simulink reference model with provided configured parameters to confirm that the module is functioning bit-to-bit accurately as specified.

The HDL of choice for the project was Very High Speed Hardware Description Language version 1993, Institute of Electrical and Electronics Engineers standard 1076-1993. VHDL was chosen for its proven reliability and manufacturability in designing Application Specific Integrated Circuits and compatibility with virtually every commercially available design and verification tool [19;11,9].

The verification language used in the project was System Verilog, IEEE standard 1800-2005 [20]. SV is a higher-level, more versatile and less verbose language to design complex test environments with. SV and VHDL can also interface with each other to an sufficient degree and passing signals from design language to another is reasonably simple to do. SV is also a widespread, standardized, well supported and documented HDL.

[11,9] The universal design methodology based test environment was written almost completely in SV [21].

The mathematical algorithm is constructed to describe adaptive FIR filtering process for noise cancellation. Matlab numerical computation based on the configured use case is to generate compatible input sources then run its theoretical test to capture output data for simulation graphical plots of the compared results. For floating-point to fixed-point conversion, Simulink reference model is generated according to the verified Matlab algorithm to be against the register transfer level description of the module during configured sample time simulation run. This model was used for generating reference data under DPI-C code format for the UVM testbench which HDL entities were binding during parallel test simulation. The output data generated by RTL was compared to the data output from the Simulink fixed-point mathematical model simulation.

The UVM test bench utilized was to verify DUT at bit level for test cycles and the electronic design automation tools used in verification management are capable of doing not only thorough code coverage analysis but also functional assertions.

2 Theory

The theoretical background for the Adaptive FIR Filtering Noise Cancellation module is presented in this chapter. The main theoretical concepts behind LMS design and brief descriptions of how to operate adaptive noise cancellation in details are introduced.

2.1 Adaptive FIR Filtering

Adaptive filters are digital filters which have a set of coefficients to be changed under step adaptations in order to reach to the optimal convergence [4,421]. Adaptive algorithm would attempt to track the statistical changes in the input signals, the optimum solution is dependent on its inherent convergence rate versus the speed of input data characteristics [1,23]. The optimization criterion to be considered is the average mean square of error difference between filter output and desired signal value. During filter coefficients adaptation, the mean square error converges step-by-step to its minimal value. This state is said to be the optimal state at which the filter output matches very closely to the desired signal. According to the optimized algorithm, the determined transfer function is controlled by variable parameters. It means the adaptive filters self-adjust with recursive algorithm to compute updated filter weights. [1,30-31.]

Normally, when the signals start from some predetermined set of initial conditions to know their priori information, it is simple to process with the fixed algorithm after a number set of iterations. But in a nonstationary environment, it no longer works as desired. [1,22-23.] Due to the special nature mentioned above, especially the tracking capability for time variation of input data, adaptive filter has been selected for practical use cases and has become an important part of DSP applications like channel equalization, echo cancellation or adaptive beamforming. The basic function comes down to the adaptive filter performing a range of different tasks, namely, system identification, inverse system identification, noise cancellation and prediction. [1,37-38.]

Although both IIR and FIR filters have been considered for adaptive filtering, the FIR filter is by far the most practical and widely used. The FIR filter has only adjustable zeros to guarantee filter stability without the need of poles adjustments as in adaptive IIR filters. That is, adaptive FIR filters are always stable. [7,595.]

2.2 Least Mean Squares Algorithm

The LMS algorithm is a stochastic gradient algorithm that it iterates each tap weight of an FIR filter in the direction of the gradient of the squared magnitude of an error signal with respect to the tap weight [1,40].

The LMS Algorithm is based on the FIR filter weights calculation. This algorithm is defined by these equations.

$$y(n) = w(n-1)u(n), \quad (1)$$

$$e(n) = d(n) - y(n), \quad (2)$$

$$w(n) = w(n-1) + \mu e(n)u(n) \quad (3)$$

Variable parameters are explained in table 1.

Table 1. Descriptions of variables from adaptive LMS algorithm

Variable	**Description**
n	The current time index
u(n)	The vector of buffered input samples at step n
w(n)	The vector of filter weight estimates at step n
y(n)	The filtered output at step n
e(n)	The estimation error at step n
d(n)	The desired response at step n
μ	The adaptation step size

The implementation of the LMS algorithm [4,436-437] is concluded by the following steps:

• Initialize w0, w1, … w(N-1) to arbitrary values where N is the FIR filter length.

• Read d(n), x(n), and perform digital filtering: $y(n)=w0.x(n)+\cdots+w(N-1).x(n-N+1)$.

• Compute the output estimation error: $e(n)=d(n)y(n)$.

• Update each filter coefficient using the LMS algorithm for k=0, ⋯, N-1: $w(k)=w(k)+2\mu e(n)x(n-k)$.

The parameter μ is the learning factor or step size that evaluates algorithm stability and its rate of convergence. It determines how fast the optimal weights are reached during step convergence. Necessary and adequate conditions of this parameter for the stability are discussed in detail in chapter 4 section 2.

For an efficient design and implementation of adaptive FIR filtering noise cancellation module under Thesis work objective, Least Mean Squares is the chosen algorithm for following reasons with its beneficial characteristics:

• It is a practical method to calculate a close estimation to optimal filter weights in real time. Accuracy is dependent on statistical sample size, since the filter weight values are based on the constant adaptive measurements of the input signals. [6,544.]

• Furthermore, its important characteristic of LMS algorithm is numerical robustness. To summarize, under the limited hardware resources, it is required to apply finite-precision arithmetic in linear adaptive filtering, the LMS algorithm generally is the preferred choice over another algorithm, for instance RLS algorithm [1,515].

Its computational complexity is simplified by a linear law that offers the ability to deliver a satisfactory in the face of unknown disturbance that can arise in practice. To be more specific, for an N-tap filter, the number of operations has been reduced to 2*N multiplications and N additions per each coefficient update. Therefore, it has a relatively simple structure in comparison to RLS algorithm and the hardware usage is directly proportional

to the number of weights. This is suitable for real-time applications and is the reason for the popularity of the LMS algorithm. [13,21.]

The LMS adaptive FIR filter architecture is as shown in figure 1 below. For every cycle, the adaptive filter automatically evaluates estimation error value for a new set of coefficients generation and computes the filter output based on applying these updates on the input data filtering.

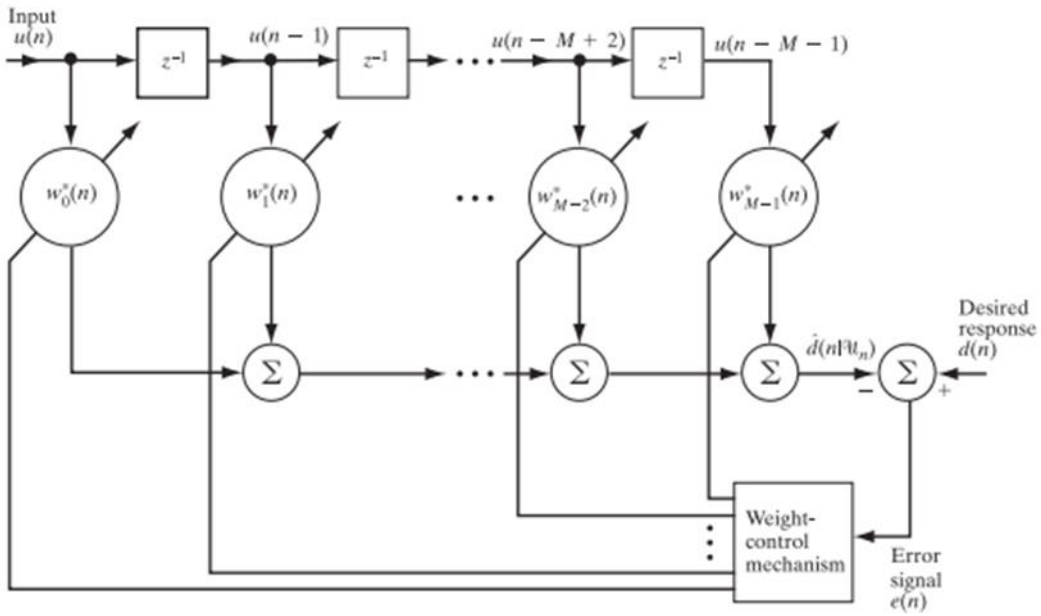

Figure 1. Structure of an adaptive LMS FIR filter

FIR filter, also referred to as a tapped-delay line filter or transversal filter, consists of three basic operations which are storage, multiplication, and addition [1,117]. These operations are described below:

The storage is represented by a cascade of M - 1 one-sample delays, with the block for each such unit labeled z-1. The number of delay elements used in the filter determines the finite duration of its impulse response. The number of delay elements, shown as M in the figure 1, is commonly referred to as the filter order. The delay elements are each identified by the unit-delay operator z-1. In particular, the delay process generates the tap inputs which represent the past values of the input. [1,117.]

The role of multiplication is that each multiplier in the filter is to multiply each compatible tap input by a filter coefficient. It forms respectively the scalar inner products of tap inputs and tap weights by using a corresponding set of multipliers. [1,117.]

At the final step, the adders in the filter is to sum the individual multiplier outputs to produce an overall response of the filter [1,117].

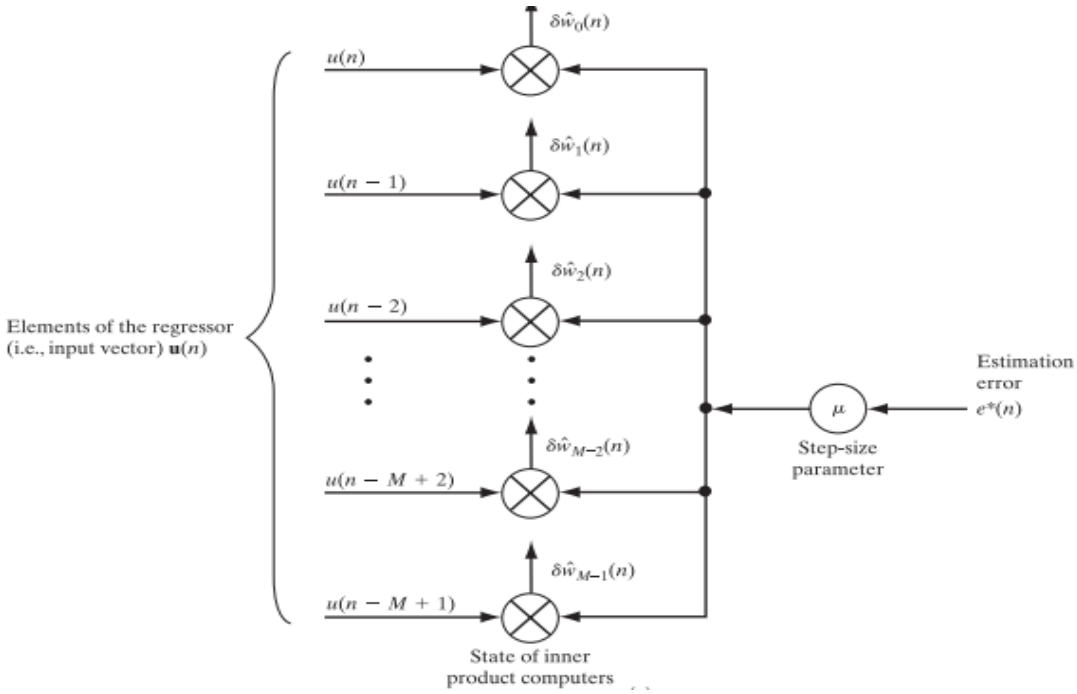

Figure 2. Detailed structure of the weight-control mechanism (Reprinted from [1,251])

For a specific analysis of adaptive weight-control mechanism, its function is to control the incremental adjustments applied to the individual tap weights of the FIR filter by exploiting information contained in the estimation error e(n). Figure 2 above presents the details of the adaptive weight-control mechanism. Specifically, a scalar version of the inner product of the estimation error e(n) and the tap input u(n - k) is computed for k = 0, 1, 2, . . . , M - 2, M - 1. The result obtained defines the correction w(n) k(n) applied to the tap weight w(n) k(n) at adaptation cycle n + 1. The scaling factor used in this computation is denoted by a positive number called the step-size parameter, which is real-valued along with the recursive computation of each tap weight in the LMS algorithm. [1,249.]

2.3 Adaptive Noise Canceller System

Noise cancellation technology is a growing field that aims to cancel or at least minimize unwanted signal. In an adaptive noise canceller to be shown in figure 3 below, two input signals are applied simultaneously to the adaptive filter. The desired signal d is the contaminated signal including the clean original signal s and the noise where they are uncorrelated with each other. The signal x, is the generated noise by measuring corrupted signal, specified to be correlated with n. This signal x is the reference input to the canceller. The signal x is processed by the digital filter according to the adaptive algorithm to self-define filter coefficients to produce an estimate y. An estimate error of the clean original signal, e is then obtained by subtracting the digital filter output, y, from the contaminated signal d. As a result, it produces the system output e expected to be the best estimate of the clean original signal s. Normally in practice, in the adaptive noise cancellation system, the clean original signal s, is the audio speech signal and the reference noise, is the signal generated from the Gaussian noise generator. [27;4,440.]

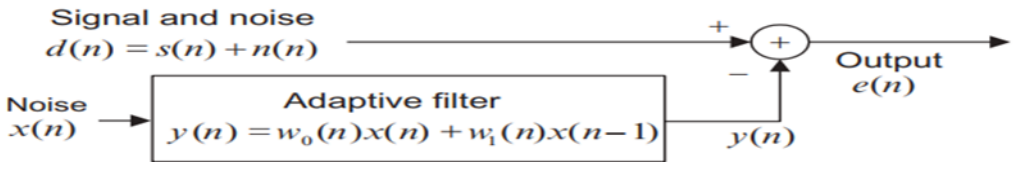

Figure 3. Noise Adaptive Canceller System Algorithm (Reprinted from [4,440])

DSP applications from adaptive noise canceller were summarized by Roger Woods [8]:

> There are many applications in adaptive noise cancellation. It can be applied to echo cancellation for both echoes caused by hybrids in the telephone networks, but also for acoustic echo cancellation in hands-free telephony. Within medical applications, noise cancellation can be used to remove contaminating signals from electrocardiograms. Adaptive beamforming is another key application and can be used for noise cancellation. The function of a typical adaptive beamformer is to suppress signals from every direction other than the desired 'look direction' by introducing deep nulls in the beam pattern in the direction of the interference. A beamformer is a spatial filter that consists of an array of antenna elements with adjustable weights. The twin purposes of an adaptive beamformer are to adaptively control the weights so as to cancel interfering signals impinging on the array from unknown directions and, at the same time, provide protection to a target signal of interest. [8,30.]

3 Digital Design Flow, Languages and Tools

This chapter introduces the implementation and verification flows, EDA tools, languages and methods used to design, implement and verify the Adaptive FIR Filtering Noise Cancellation module. Digital design and implementation is presented first before logically moving forward onto the verification stage. Firstly, the digital design flow is introduced and the brief overview for each fundamental step is discussed. In addition, the hardware description language chosen for this project design and verification is mentioned. Lastly, the EDA tools required to be used in this project are demonstrated with functional features. In the verification sections, the verification flow is presented to show its important connections with the digital design flow. With the same description as in digital design sections, verification method and language utilized in this project are also mentioned with relevant features to be compatible with EDA tools.

3.1 Digital Design Flow

The design module is implemented according to Mathworks workflow [26]. The mathematical algorithm and functional specifications are prepared with a set of design constraints at the initial design phase. The functional specification outlined the range of capabilities and features the module should follow. The mathematical algorithm is experimented in detail under floating point implementation then it is converted to fixed-point model as a reference modular and configurable code which will be developed to create a Simulink golden model. Furthermore, there are Register Transfer Level implementations of corresponding Simulink components. A typical top-down digital design flow shown below in figure 4. The flow also comprises some feedback paths if needed for design iterations to be effectively adjusted at design stages. It aims to create module compatibilities between design algorithm and RTL implementation. One of the iterative feedback loops was in between fixed-point conversion step and algorithm functional specifications or between logic design constraints and Simulink reference model implementation. Fixed-point mathematical algorithm is the reference algorithm which would be verified to be in comparison with its floating-point implementation under the allowed predefined tolerance limitation. If there is a bad efficiency, based on available limited resources, for instance the maximum signal word length, the algorithm functional specifications at the early design phase must be reconfigured. Besides, RTL implementation

usually need a few design iterations before the Simulink model can be successfully realized in RTL because some Simulink constructs are not even feasible to implement synthesizable RTL design, leading to necessary adjustments.

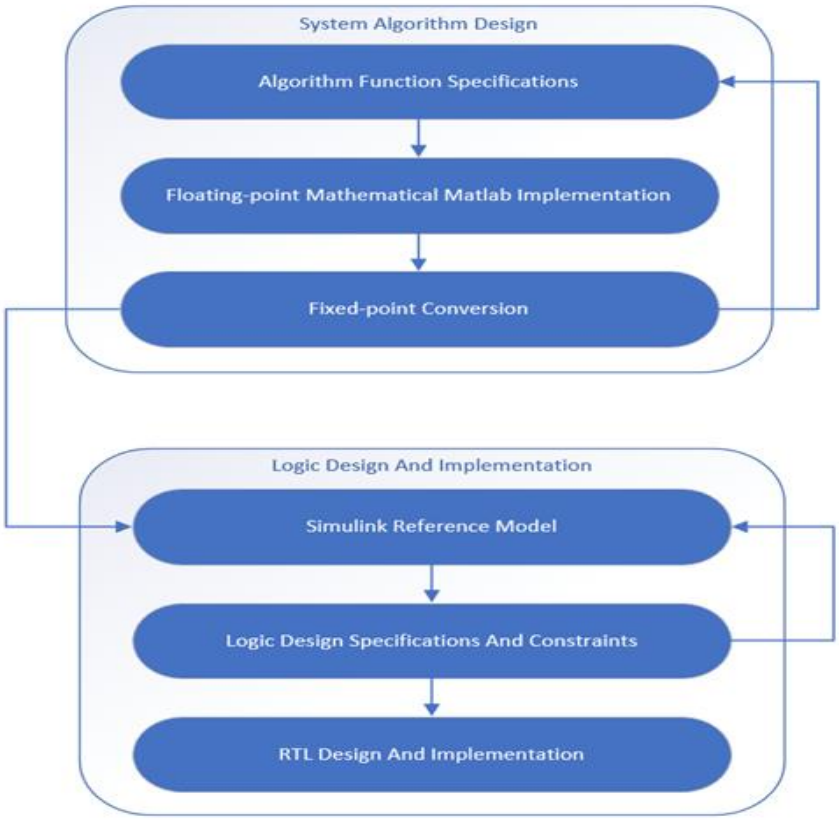

Figure 4. Digital logic design and implementation flow (Drawn by using Microsoft Visio)

Digital design flow consists of system analysis, design and verification of a system reference model, coding and verifying RTL description of the reference model.

System analysis is an early design phase, which includes system functional specifications, architecture specifications, coding the mathematical algorithms of the system. Architecture and algorithm choices during the system analysis affect system development and may have a great effect on design complexity [12,9].

The Simulink reference model to be developed from the mathematical algorithm describes the behavior of a system. RTL, which is hand-written based on the reference Simulink model, use HDL simulator to run the input data through HDL entities that these

data will be in comparison with output data generated by Simulink simulation for each bit cycle. RTL design codes are used to demonstrate hardware functionality.

3.2 Hardware Design and Verification Languages

3.2.1 VHDL for RTL Design

Very High Speed Integrated Circuit Hardware Design Language was originally developed by the order of the U.S Department of Defense (DoD) in the early 1980s. The purpose of VHDL was to document the behavior of the ASICs that were supplied to the DoD included in the equipment from their subcontractors. The next logical step was to develop a simulator capable of reading the created VHDL documents. Since the VHDL documents contained the exact behavioral model of the ASIC, logic synthesis tools were developed so that the HDL code could be turned into the definition of the physical implementation of the circuit. [11,36.] The Adaptive FIR Filtering Noise Cancellation module designed was written completely in VHDL.

3.2.2 System Verilog for Verification

System Verilog is a hardware design and verification language that is an extension to Verilog HDL. The first IEEE standard, 1800-2005, of System Verilog was based on Superlog design language and Open Vera verification language which were donated to the System Verilog project by Accellera and Synopsys respectively. In 2009 System Verilog was merged with base Verilog creating the SV IEEE 1800-2009 standard. The current standard of System Verilog is IEEE 1800-2012. [11,46-47.]

Test benches and verifying in general are not bound to the limitations of physical circuit design and because of this SV for verification has adopted many techniques from traditional programming languages such as C/C++, where SV adapted its syntax and Java [11,47]. SV for example supports object-oriented programming (OOP) techniques where objects are manipulated instead of data [20]. OOP is yet to make a breakthrough in HDLs but in test environments it is convenient to consider the different parts of the testbench as separate interconnected objects whose properties can be described and altered to suit the current verification task at hand. In SV objects are instances of classes which can inherit all or part of the properties of the class they are inherited from. OOP is a very useful programming paradigm for test bench since the code produced is very flexible, reusable and transferable with little extra effort. [11,47.]

The verification environment used to verify the design module is written almost completely in System Verilog with DPI-C generated components and is based on UVM.

3.3 Verification Flow and Universal Verification Methodology

The verification flow applied for the Thesis project is shown in figure 5. At the initial design stage, design specifications are considered with a verification plan for feasibility. After this period, verifications are implemented to guarantee the module functional characteristics to be expected. That is, floating-point code and fixed-point mathematical code are used to verify the module functional characteristics. If any design error occurs during simulation, there would be an inevitable change on algorithm specifications. Simulink modeling is used to be as a reference algorithm due to its characteristics for visualization and sample-based simulation. That is, the design dataflow can be observed sample by sample during the simulation for fast error checking and easy debugging ability to intervene and adjust block properties to achieve model functional correctness. Configured diagnostics for the Simulink reference model are observed after simulation experiments to check model functionalities. In addition, Simulink components of each subsystem are utilized to be converted to DPI-C reference mathematic coding which is run under UVM test environment through Direct Programming Interface to generate the reference input stimuli and expected output data. These stimuli are driven into RTL DUT and RTL captured output will be in comparison with the available expected output from DPI-C coding parallel simulation. For RTL verification, HDL testbench is coded to verify RTL design in the specific use case to meet the functional specifications. Logic simulation is applied by utilizing this testbench. Later, to debug whether there is any design misoperation which differentiates RTL design from the reference model or not, HDL Cosimulation is used to set up mutual connections between Simulink and HDL Simulator. The identical stimuli are applied to both Simulink reference model and RTL DUT for parallel simulation then their outputs are captured to be compared with design goal which is to achieve the bit-to-bit zero difference.

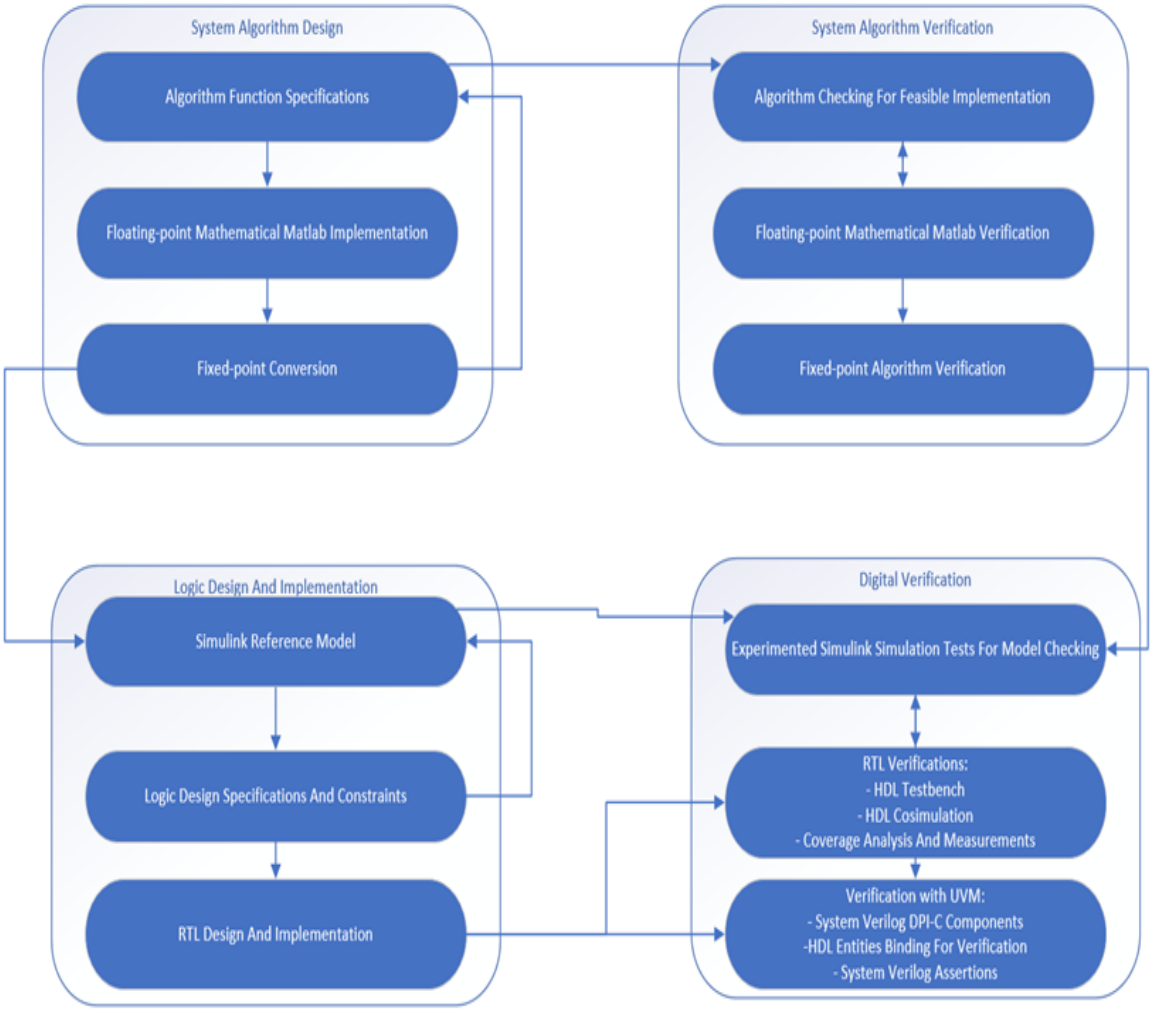

Figure 5. Digital verification flow and its relation to design flow (Drawn by using Microsoft Visio)

After passing these verifications, UVM is the higher-level verification method to be applied with System Verilog programming that engineers could have used for decades due to its configurable code features based on Verilog constructs.

Traditionally, the verification of HDL programs has been done in the same language the design was written. A testbench was designed for the design under test (DUT) and test stimulus was applied to the inputs of the DUT (HDL program) to check if the module functions correctly and the output is what was expected. This method of verification however is both time consuming and unreliable since every test case must be hand-crafted and exhaustively testing a big design is quite a task. Several verification methods were developed to tackle this issue and after a long period of vendor specific verification methods, the Universal Verification Methodology (UVM) was born. UVM is a standardized

methodology now supported by all the major simulator vendors. The UVM class library brings a lot of automation to the SystemVerilog language such as sequences, data automation features and macros. With UVM it is possible to create reusable, automated and adaptable test environments that accelerate the verification of complex hardware designs and make it possible to achieve much better test coverage with reduced effort compared to the traditional testbench-design approach. [20;21;11,46.]

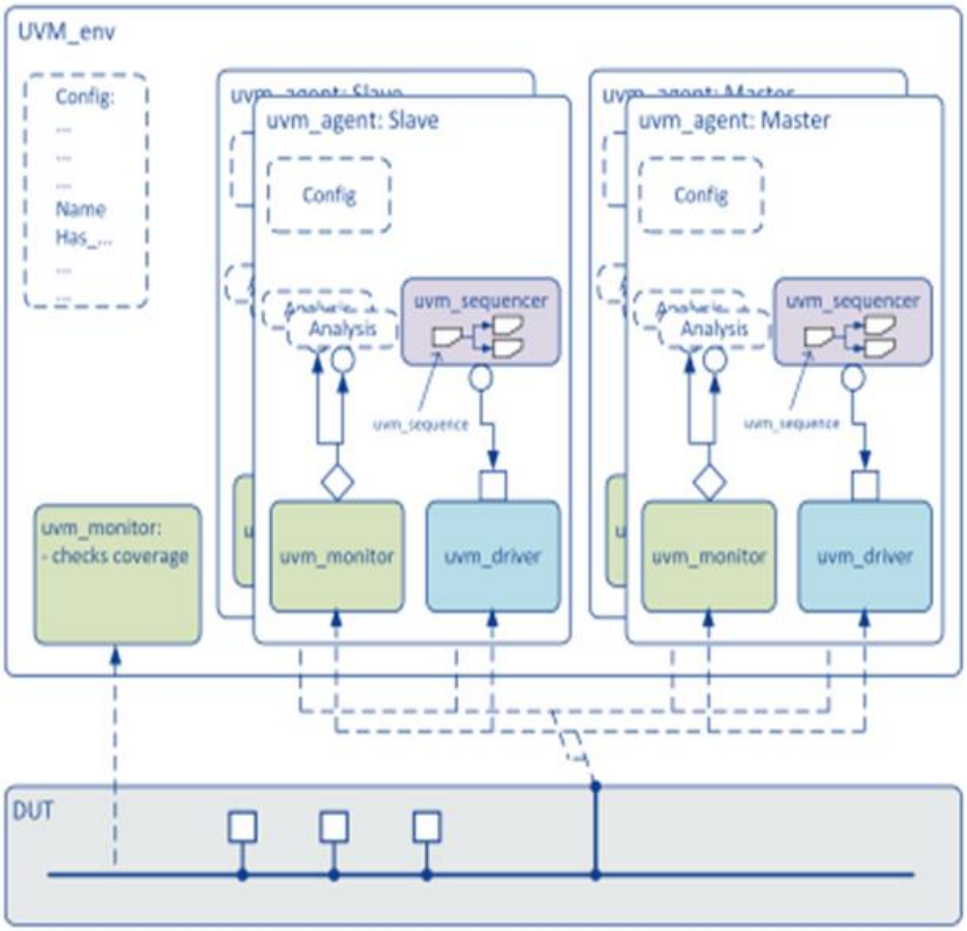

Figure 6. Verification Component Environment with UVM library classes (Reprinted from [11,53])

Verification components such as sequencers, monitors, drivers and agents are built using the blocks found from the UVM class library. The UVM classes from the UVM Class Library provide the building blocks needed to quickly develop well-constructed and reusable verification components and test environments in System Verilog as presented above in figure 6. The classes contain verification specific features for easier building of verification components. The UVM Class Library also provides utilities for debugging with user-controlled messages that can be used, for example, to report failures and the status

of every test operation on any level of the testbench, even globally. To be more precise, the UVM classes and utilities are divided into categories pertaining to the specific roles or functions such as UVM core base class, reporting classes provide a facility for issuing reports with consistent formatting and configurable side effects based on design verbosity and severity, factory overrides to manufacture UVM objects and components, phasing capability, configuration database, type parameterized data structures, and UVM reusable and configurable verification components. [21;11,53.]

Below are the main UVM verification components [11,53-54] explained briefly:

• Data item: Any input to the DUT network packets, buses, transactions or protocol values and attributes.

• Driver: Emulates the logic driving the DUT. Accepts data input which it can sample and feed to DUT.

• Sequencer: Stimulus generator that feeds the driver.

• Monitor: A passive entity that samples DUT signals. Monitors collect coverage information and perform the checking of the data.

• Agent: An agent encapsulates drivers, monitors and sequencers to simplify the usage of verification components by providing an abstract container which can perform a complex task without independently configuring every component.

• Environment: Top-level component of verification components and agents has the configuration properties for customization and re-use of the whole component topology without altering the internal components.

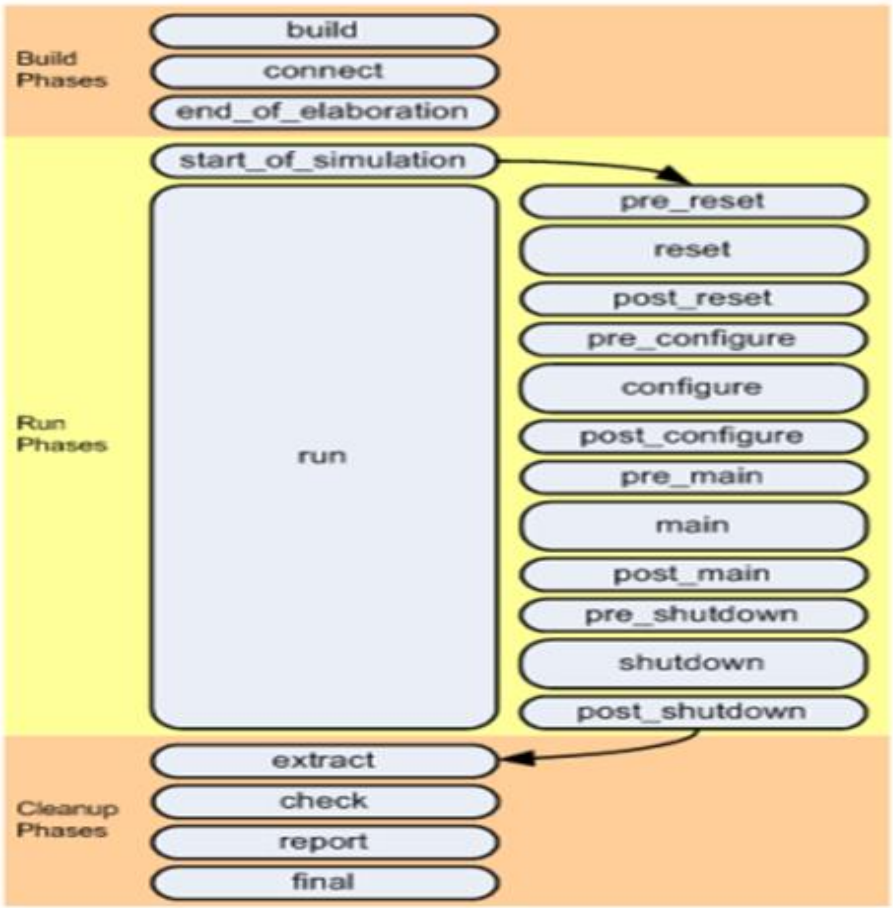

Figure 7. The UVM Phases (Reprinted from [17,11])

The UVM uses phases which are described in detail in figure 7. It allows a consistent testbench execution flow during simulation. There are three different groups of phases executed in the following order [17,11.]:

• Build phases: where the testbench is configured and constructed.

• Run-Time phases: where time is consumed in running the testcase.

• Clean-up phases: where the results of the testcase are collected and reported.

4 Design Specifications and Algorithm to Architecture

The Adaptive FIR Filtering Noise Cancellation module updates the filter weights based on LMS algorithm as FIR filtering procedure. The original input signal source is a complex-valued sinusoidal signal which is stimulated with configured length. The noise source is randomly generated then being filtered by FIR filtering process to create the corrupted noise. After the addition of the original input and the corrupted noise, the result of this procedure is the desired signal source which will be scaled for fixed-point implementation.

Before adaptive FIR filtering implementation, through configured tests, adaptation step size is selected to optimize the adaptive filtering process.

The tapped delay noise input and the estimation error output after each adaptive filtering procedure are used to calculate the updated filter weights based on adaptation step size.

4.1 LMS Algorithm for Digital Design and Implementation

Based on the diagram as in figure 8, LMS algorithm consists of two basic processes that continually build mutually:

• Filtering process: filter output computation includes FIR filter design of noise input based on updated weights to get the calculated filtered signal and the subtraction of this filtered signal from the desired input signal to obtain the estimation error output.

• Adaptation process consists of weight update logic, tapped delay input processing, weight update controller.

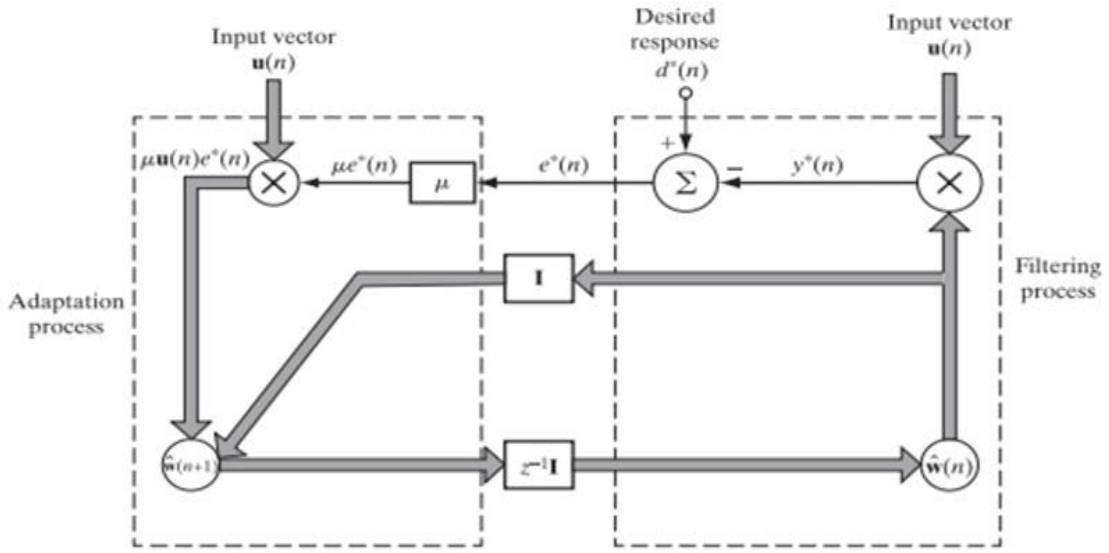

Figure 8. Signal-flow graph representation of the LMS algorithm (Reprinted from [1,267])

The specified architecture is required to perform a design bit-level architecture for vector-vector multiplication used for output calculation with finite convolution sum applied algorithm.

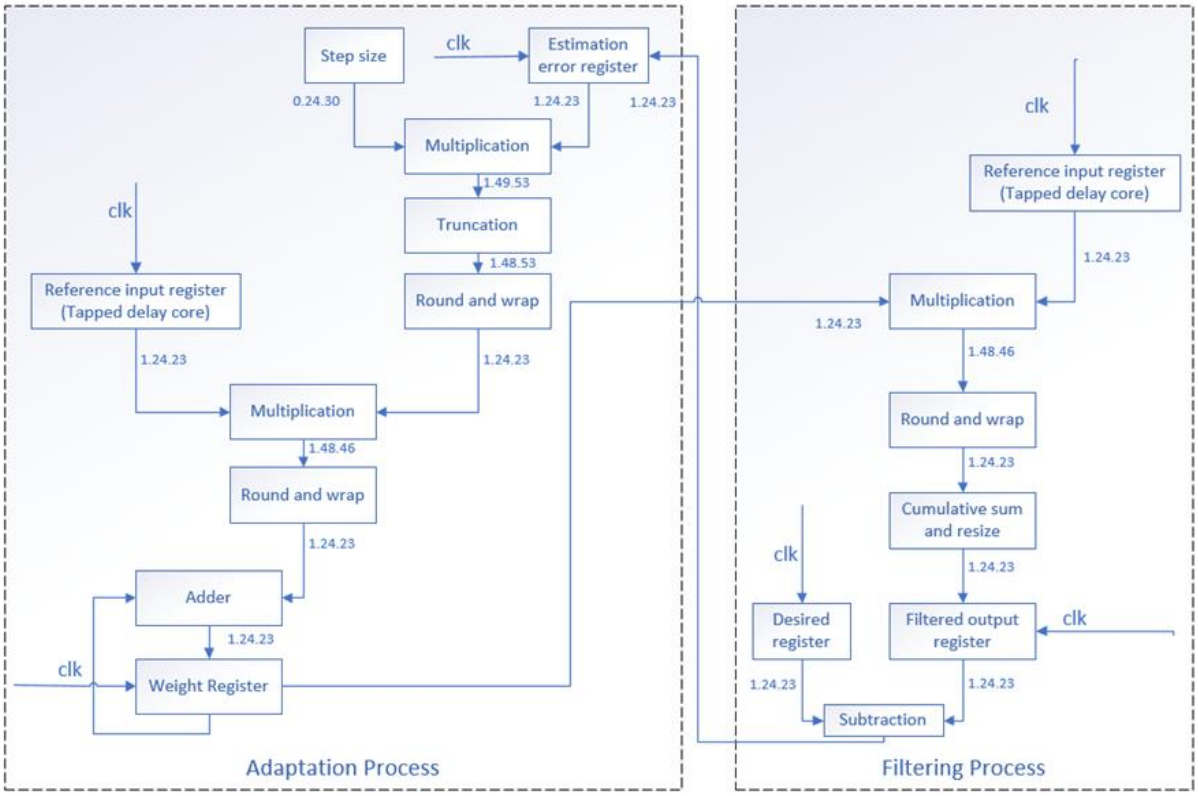

Figure 9. Detailed RTL architecture of LMS algorithm (Drawn by using Microsoft Visio)

A detailed top-level module architecture is demonstrated in figure 9. It contains two partial designs: filter weights adaptation and filter output computation. The left-side design describes how the filter coefficients adapt to the optimal solution. For each valid clock cycle, the delayed input is generated from Tapped Delay core. The estimation error register will multiply by the fixed step size parameter, then the result will go through truncation, round and wrap procedures respectively. This result called adaptive correction will continue to multiply by the delayed input before rounding and wrapping to get the needed offset for weights adaptation. The addition between the weight offset and the weight register is to update to create a new set of filter coefficients. Meanwhile, filtering process is parallelly operated, which can be seen in detail at the right side of the figure 9. It describes the filter output computation. To be more specific, for each valid clock cycle, there occurs the multiplication between the generated set of filter weights and the delayed input to result in the output array with its size as the filter length order. After round and wrap, cumulative sum of elements and resize procedures, the final filtered output is used for subtraction operation based on noise cancellation algorithm to obtain the estimation error register. The Q-format [10,530] to be shown for each signal are configured in advance, is 1.24.23 that it means the signal is represented in signed binary format, 24 is the word length and 23 is the fractional length. In summary, these two design processes operate simultaneously and have a mutual impact on each other.

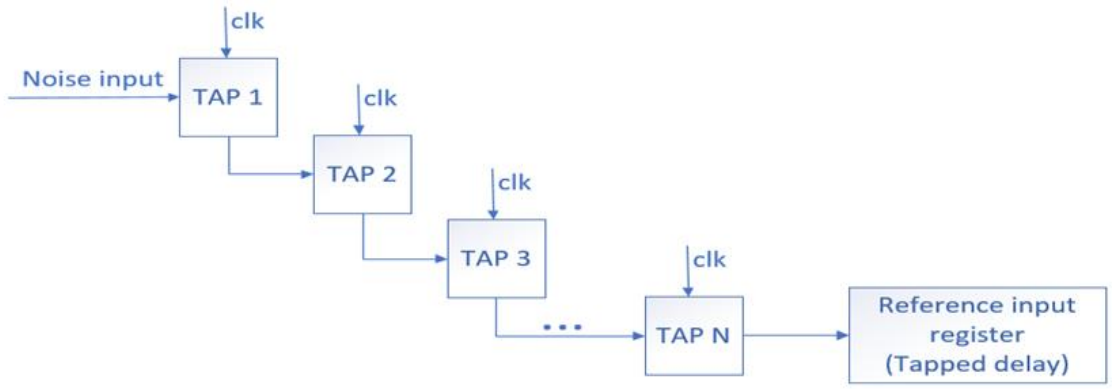

Figure 10. Tap delay processing of RTL Tapped Delay core (Drawn by using Microsoft Visio)

The Tapped Delay core for the adaptive noise cancellation module is shown in figure 10. In the tap processing, each tap is controlled by the valid clock signal. The input from each tap is shifted and stored to be combined with the next tap input after one-cycle delay operation. The reference noise input is propagated through all taps and automatically replaced by shift arithmetic process. [27.]

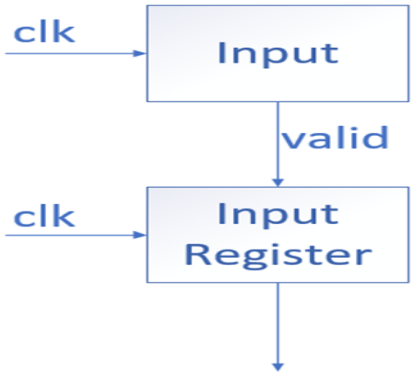

Figure 11. Weight update control mechanism for RTL (Drawn by using Microsoft Visio)

Weight update control mechanism is shown in figure 11. For every rising edge of the clock signal, the weight update process evaluates the valid signal which enables the register operation for signals to be used for mathematical computations.

4.2 Design Configurations and Constraints

The LMS algorithm designed for adaptive FIR filtering noise cancellation module is complex in the sense that the input and output data as well as the tap weights are all complex valued.

To be more specific, Matlab implementation, floating-point to fixed-point conversion, coding style and verification functional requirements are the considered aspects at the early design period to specify detailed variable parameters for design compatibility as well as its configurability.

4.2.1 Floating-point Matlab Implementation

Relevant parameters related to adaptive LMS FIR filtering algorithm is configured as below:

• FIR filter length is 45, which is not only sufficient for coefficient adaptation but also an effective tap order number for FIR filtering.

• Stimulus length is 1000, which is high enough to allow an effective design functional characteristics observation.

• Scale Factor Adjustment: It is acceptable for accuracy limitation when transitioning from a floating point to a fixed-point representation. The conversion method is to keep a limited number of decimal digits. Normally, two to three decimal places method is appropriate for digital filter processing. Therefore, the scale factor can easily be obtained by multiplying the data values by 100 and divide the calculated values by 100 to recover the anticipated value. [13,24-25.] For complex signal, the common scale factor is the maximum value between the calculated scale factor of real part and the another one of imaginary part to guarantee that the output values are in the interval [-1, 1].

• Clean original input signals is a complex-valued sinusoidal signal which includes in dependent sinewave sequences containing the sine of elements over the domain which ranges are proportional to (−π,π) with multiplying constants of 5 and 6 respectively. Their step-incremental values are also in the interval [-1, 1] with a determined range which is corresponding to the configured stimulus length. Therefore, signal resolution is about 6.23*10^^-3 (radians).

• The noise input is generated as a random Gaussian noise sequence to be scaled by dividing by the common scale factor. The correlated noise input is generated by using a 22-th order window-based FIR bandpass filter with passband 0.35π≤ω≤0.65π rad/sample. The desired input is an addition of the clean sinewave input and the correlated noise input after being scaled by dividing by the common scale factor.

• The necessary condition for the step size parameter u to satisfy the convergence time constant is 0 < u << 1 [3,296]. In addition, if u is made too large then the algorithm becomes unstable. Therefore, u is determined to be optimal by test experiments using Matlab coding with pre-defined conditions.

4.2.2 Floating-point to Fixed-point Conversion

Floating point circuits have higher power consumption and lower performance compared to fixed point. For the vast majority of DSP applications, fixed point arithmetic is suitable, especially FIR filtering algorithm. [9,545-546.] For fixed-point conversion, signal dynamic range is mapped into the limited fixed-point precision [2,366].

Using finite word lengths prevents us from representing values with infinite precision, inaccurate arithmetic results. The level of output roundoff noise in fixed-point implementations can be reduced by increasing the word length. [5,647.]

The value bit width of 24 is chosen to be configured for quantization process due to its step resolution of 1/8388608 which works out to 1.192 * 10-7, leads to the significant reduction of error percent of signal level. To perform mathematic computations in RTL design, Q1.23 format to represent signed values with fraction bits of 23 is used.

The next finite word-length effect to be considered is data overflow. Round and wrap on overflow are the chosen method due to hardware resource accuracy minimization, especially suitable for configured bit-level design. For addition and multiplication processes in RTL coding, the results are always truncated to Q1.23 format to meet the data bit width requirements as configured before.

4.2.3 Coding Style

MATLAB is generally used for algorithm design of a system for fast simulation and verification purposes of the behavioral model. To produce rational HDL code, the algorithm should be written from the hardware perspective. Algorithm models are often written into simulation optimized vector operations that create parallel structures and copies of combinational logic blocks in hardware. To create synthesizable MATLAB code, the structure must be correct. Register modeling is done through “persistent” variables. The variables that are wanted to save their states are defined in MATLAB function as persistent and these variables generate registers into RTL. [12,21-23.]

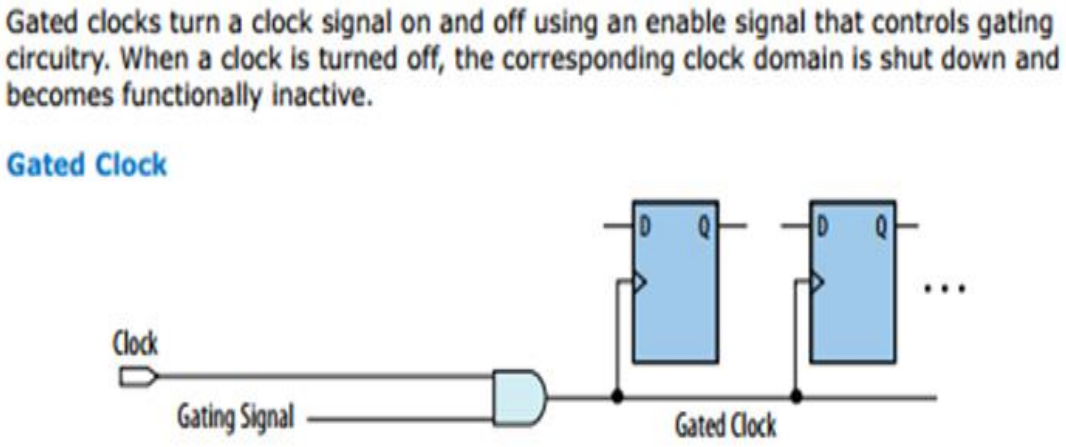

Figure 12. Gated clock method to reduce power consumption (Reprinted from [22,12])

RTL implementation uses gated clock method as described in detail in figure 12 is utilized to reduce power consumption in device architectures by effectively shutting down portions of a digital circuit when not being in use [22,12].

4.2.4 Verification Requirements

All-passed verification from simulation-based verification with HDL testbench, bit-accurate parallel verifications with HDL Cosimulation and most importantly, UVM testbench that use constrained stimulus generation and functional coverage methodologies is the must-meet design goal. Apart from modular and reusable coding style, specifically for the mathematical algorithm, Matlab implemented coding is re-quired to be in separate functions which are targeted to a dynamic system.

In addition, the structural coverage analysis with code coverage result of 100% along with all concurrent System Verilog assertions evaluated to be passed are the fundamental factors to clearly demonstrate design functional correctness.

5 Adaptive FIR Filtering Noise Cancellation Module Implementation

5.1 Matlab Analysis and Implementation

5.1.1 Floating-point Matlab Implementation

When the LMS algorithm is operating in a limited-precision environment, the point to note is that the step size parameter may be decreased only to a level at which the degrading impacts of round-off noises in the tap weights of the finite-precision LMS algorithm become significant [1,515].

In addition, as mentioned in section 4.2.1, step size that is too small increases the time for filter convergence or that is too large may cause the adaptive filter to diverge [27]. Therefore, to be in compliance with the step size condition, the step size u must be 0 < u << 1.

```
coeff_len = 45;
mu = 0.008% Set the step size for algorithm updating.

lms = dsp.LMSFilter(coeff_len,'Method','LMS',...
   'StepSize',mu,'InitialConditions',0);
[~,e] = lms(noise,d);
L = 1000;
plot(0:999,signal(1:1000),0:999,e(1:1000));
title('Noise cancellation performance by the LMS algorithm');
legend('Actual original signal','Filtered error signal after noise reduction',...
       'Location','NorthEast')
xlabel('Time index')
ylabel('Signal value')
```

Listing 1. Matlab implementation for adaptation step size test selection

As seen in listing 1, the design configurations are fixed as specified in section 4.2.1 for input signals, the pre-defined filter length of 45 taps and the stimulus length of 1000. The available Matlab supported system object dsp.LMSFilter is utilized to compute estimation error output of the adaptive LMS filter according to the step size parameter which is experimentally determined. XY plot is used to show the output signals over time.

In figure 13, the blue signal is the clean original sinusoidal wave which runs at the same time with the filtered output signal which is the red one. The optimal solution is reached when the error difference between these two signals is at the minimum value. Therefore,

after test experiments with gradual reduction of step size parameter, 0.008 is the best choice because it is small enough but efficient to meet the design requirements to get the optimal convergence.

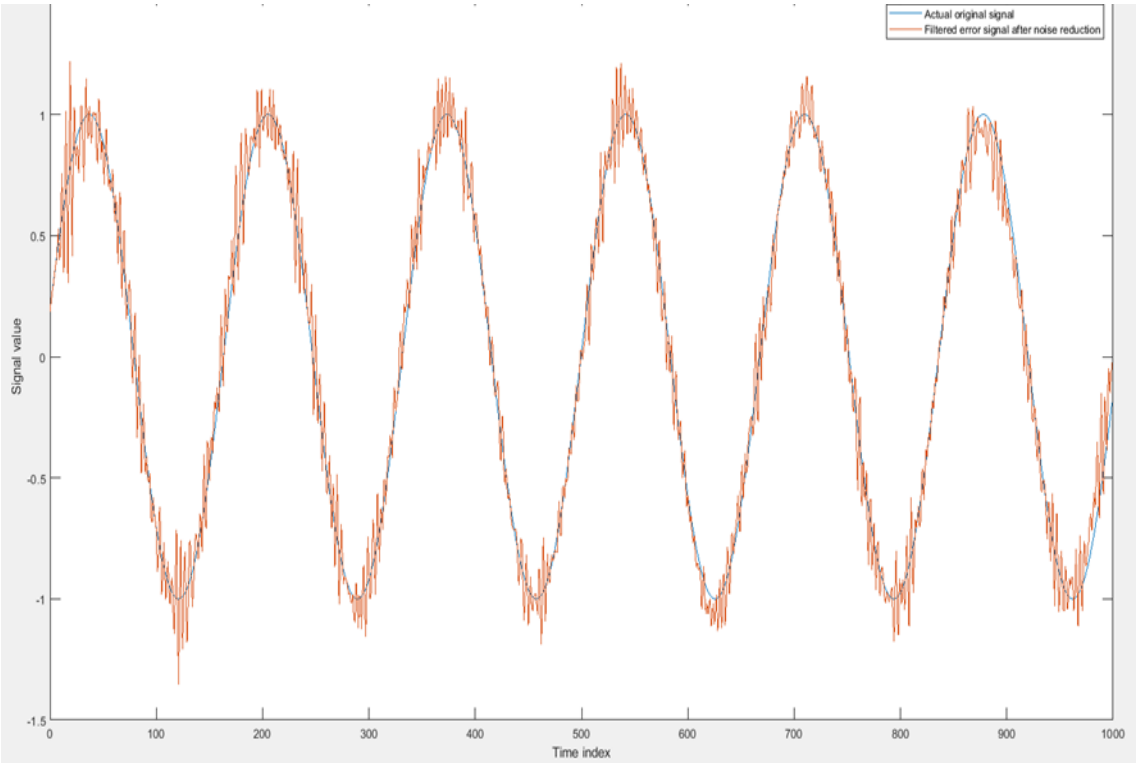

Figure 13. MATLAB GUI displaying the result of step size test selection after simulation

After defining the fixed adaptation step size, other design specifications must be configured independently as shown in listing 2. That means it is easy and convenient to modify if there is any change from the design algorithm. Matlab top-level design and Simulink reference model import this configuration as a design callback.

```
function config = init_config()
config.simtime = 10;
config.stim_len= 1000;
config.reset_weights =false;
config.update_weights = true;
config.coeff_len = 45;
config.del_num = 44;
config.step_size = 0.008;

end
```

Listing 2. Matlab function for initial design configurations

The detailed declaration for reference inputs of the Matlab mathematical testbench are shown in listing 3. These input declarations are to follow the design specification specified in section 4.2.1. The two inputs, which are the reference noise and the desired signal, are both generated in the sense of complex values. Moreover, window-based FIR filter design with the usage of dsp.FIRFilter system object is to generate correlated noise from original noise input. It uses a Hamming window to design a 22-th order FIR bandpass filter with passband $0.35\pi \leq \omega \leq 0.65\pi$ rad/sample. The correlated input will be added to the clean sinewave to form the desired signal. Obviously, the scaling method is applied for original noise input and desired signal to adjust their value ranges to be in the interval [-1, 1].

```
% Create filter structure to generate correlated noise from noise input
filt2 = dsp.FIRFilter(...
        'Numerator', firl(22, [.35, .65]));
rng('default');
x = complex(randn(stim_len,1),randn(stim_len,1)); % Generate the noise input
% Scaling the noise input to +/- one
scale_noise = max([abs(ceil(real(x)*100)/100); abs(ceil(imag(x)*100)/100)]);
x = x/scale_noise;
% Generate the original input signal ( sinewave/random sequence)
w = [-0.99*pi:1.98*pi/(stim_len-1):0.99*pi] ;
stimulus = complex(transpose(sin(6*w)),transpose(sin(5*w)));
d = step(filt2,x) + stimulus;          % Correlated Noise + Input Signal

% Scaling the desired signal to +/- one
scale_signal = max([abs(ceil(real(d)*100)/100); abs(ceil(imag(d)*100)/100)]);
d = d/scale_signal;

noise_in = x ;
desired_in = d ;
```

Listing 3. Input declarations for adaptive FIR filtering noise cancellation testbench

Down to the verification of the Matlab testbench as can be observed in listing 4, it calls the main implementation of adaptive LMS FIR filtering noise cancellation for separate data paths. Because both of two inputs are complex-valued format, to implement the adaptive LMS algorithm, their data paths are divided into real path and imaginary path. Each data path is independent to implement the adaptation process before complex-valued combination only at the final stage to get the final outputs. Therefore, there are two separate comparison plots for each data path.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Call to the design
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
for i = 1:stimulus_length
    [y_re(i),err_re(i)]=lms_fcn_re(x_re(i), d_re(i), stepSize, reset_weights,update_weights,coeff_len);
    [y_im(i),err_im(i)]=lms_fcn_im(x_im(i), d_im(i), stepSize, reset_weights,update_weights,coeff_len);
end

figure('Name', [mfilename, '_signal_plot']);
subplot(5,1,1), plot(stim_re), title('Original Stimulus Input Signal (Real part)');
subplot(5,1,2), plot(x_re), title('Original Noise Input Signal (Real part)');
subplot(5,1,3), plot(d_re), title('Desired Signal (Real part)');
subplot(5,1,4), plot(y_re), title('Filtered Signal (Real part)');
subplot(5,1,5),plot(err_re), title('Error Signal (Real part)');

figure('Name', [mfilename, '_signal_plot']);
subplot(5,1,1), plot(stim_im), title('Original Stimulus Input Signal (Imaginary part)');
subplot(5,1,2), plot(x_im), title('Original Noise Input Signal (Imaginary part)');
subplot(5,1,3), plot(d_im), title('Desired Signal (Imaginary part)');
subplot(5,1,4), plot(y_im), title('Filtered Signal (Imaginary part)');
subplot(5,1,5),plot(err_im), title('Error Signal (Imaginary part)');
```

Listing 4. Call to the main design implementation from Matlab testbench and graphical plots of outputs after test simulation

The design goal is to implement design code to be configurable and reusable that there are modular functions coded for partial designs. Besides, to register the filter coefficients for Matlab mathematical algorithm, persistent variable declaration is used as can be seen in listing 5.

```
% register filter coefficients
persistent filter_coeff;
if isempty(filter_coeff)
    filter_coeff = zeros(1,coeff_len);
end
```

Listing 5. Persistent variable usage for RTL description of filter weights

Adaptive LMS FIR filtering noise cancellation implementation is mainly divided into three modular functions to be shown in detail in listing 6, listing 7 and listing 8. These functions aim to be the design callbacks when the top-level design call them to perform FIR filtering and weight update process before output computation at the end of each step adaptation.

```
function tap_delay = tapped_delay_re(input,coeff_len)
% The Tapped Delay function delays its input by the specified number
% of sample periods, and outputs all the delayed versions in a vector
% form. The output includes current input


persistent u_d;
if isempty(u_d)
    u_d = zeros(1,coeff_len);
end
a=zeros(1,coeff_len);
for i=1:(coeff_len-1)
    if i == (coeff_len-1)
        a(i) = u_d(i+1);
        a(i+1) = input;
    else
        a(i)= u_d(i+1);
    end
end
tap_delay = a;
u_d(:) = a ;


end
```

Listing 6. Tapped delay Matlab function

As seen in listing 6 above, the noise reference input is delayed at each simulation step to form 45 taps which is previously defined as exactly equal to the filter coefficient length. The persistent variable u_d is responsible to store the tap input which is in vector form for shifting process when the top-level design calls the tapped delay function.

```
function weights = update_coeff_fcn_im(step_size, error_sig, delayed_signal, filter_coeff, reset_weights,update_weights,coeff_len)
% This function updates the adaptive filter weights based on LMS algorithm
step_sig = step_size .* error_sig;
adaptive_correction = delayed_signal .* step_sig;
updated_weight = adaptive_correction + filter_coeff;
if reset_weights
    weights = zeros(1,coeff_len);
elseif     update_weights
    weights = updated_weight(1:coeff_len);
else
    weights = filter_coeff;
end
```

Listing 7. Weight update logic Matlab function

The main adaptation procedure is described in listing 7. The weight update logic is implemented step-by-step in a clear coding style, follows the LMS reference algo-rithm. The Matlab code is to use reset_weights variable and update_weights variable to set the weight update control mechanism.

```
function output = cusum_re(input)
output = 0 ;
for i = 1:numel(input)
    output(:) = output + input(i) ;
end
end
```

Listing 8. Cumulative sum of elements Matlab function

FIR Filtering process is simply described in listing 9. To be more specific, after vector-vector multiplication between tapped delay input and the filter coefficient set, cumulative sum of elements procedure shown in detail in listing 8 is implemented to calculate the filtered signal based on the computed filter weights as the input variable.

```
% Create filtered signal using FIR filter algorithm
weight = delayed_signal .* filter_coeff;
filtered_signal = cusum_re(weight);
```

Listing 9. Filtered output computation procedure from Matlab reference design

After design steps of Matlab mathematical algorithm mentioned previously, the Matlab testbench is run to verify adaptive FIR filtering noise cancellation reference design. The plot of outputs for each data path is shown in figure 14.

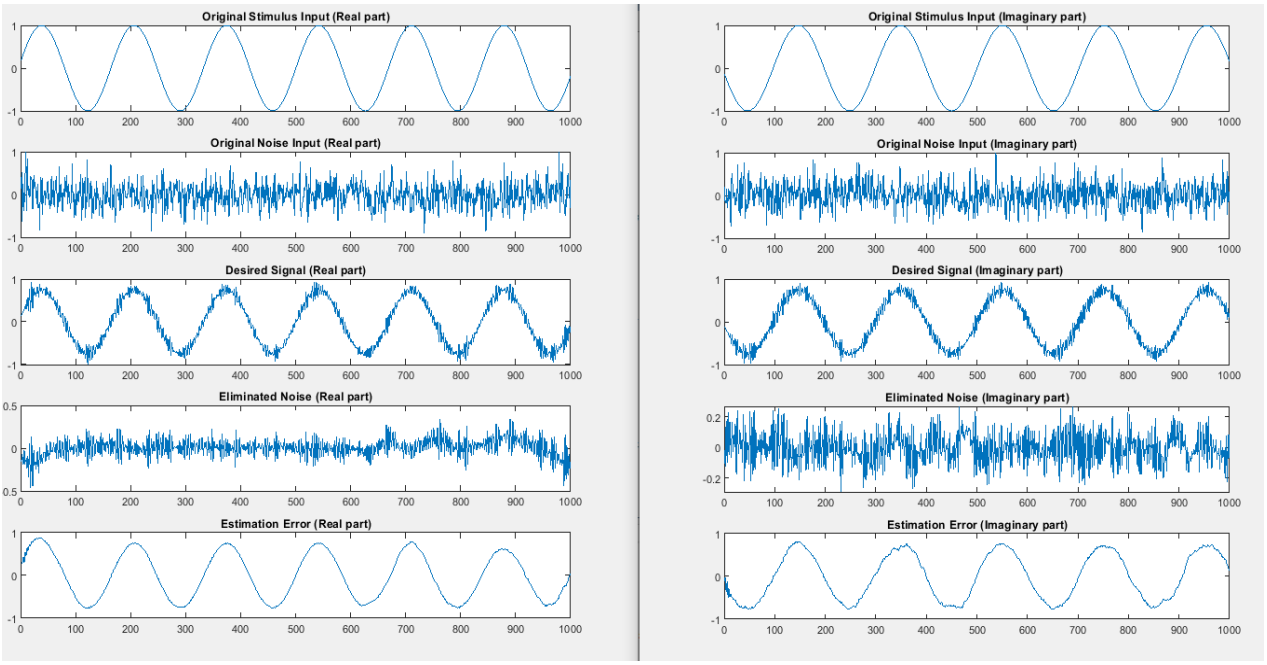

Figure 14. Graphical plot to show Matlab testbench outputs after simulation

The adaptive FIR filtering noise cancellation is successfully operated. The desired signal is the addition of the clean sinewave stimulus and the noise reference input. To be compared with the desired signal, the estimation error signal reached the cleaner version due to noise elimination. To be more precise, the estimation error is the result of the subtraction the FIR-filtering filtered output signal from the desired signal.

5.1.2 Fixed-point Matlab Mathematical Testbench

To implement floating-point to fixed-point conversion, fixed-point mathematics configuration is simply specified in Listing 10 below which is added to every Matlab floating-point code.

```
fim = fimath('RoundMode','round','OverflowMode','Wrap','ProductMode','FullPrecision','SumMode','FullPrecision');
dat = numerictype(1,24,23);
step_v = numerictype(0,24,30);
```

Listing 10. Matlab fixed-point mathematics configuration

As listing 10 demonstrated, step_v is exceptional to be configure only for fixed-point step size while dat is configured for remaining signals.

5.2 Simulink Reference Modelling

Simulink is a graphical design tool that uses library blocks, MATLAB functions and System Objects to perform indicated tasks and functions. Simulink library contains huge selection of hardware optimized blocks. These blocks are divided into different categories to be integrated in SoC development. User-defined MATLAB function blocks and System Object blocks can be used to self-define design specific configurations or to reutilize available MATLAB mathematical computations. The blocks are built in Simulink modeling by referring the Matlab reference algorithm with fixed-point data types. A test bench in Simulink is built by driving all input variables into Device Under Test and capturing DUT outputs to be verified. The data is imported as streaming data with generic parameter configurations. [12,24-25.]

Simulink Reference Model implementation is based on Matlab fixed-point design algorithm. The top-level model is divided into three main parts shown in figure 15. The data flow goes left to right from sequence generation, main adaptive filtering design to model checker. Each module subsystem is designed in detail that can be seen in figure 16, figure 18, and figure 19.

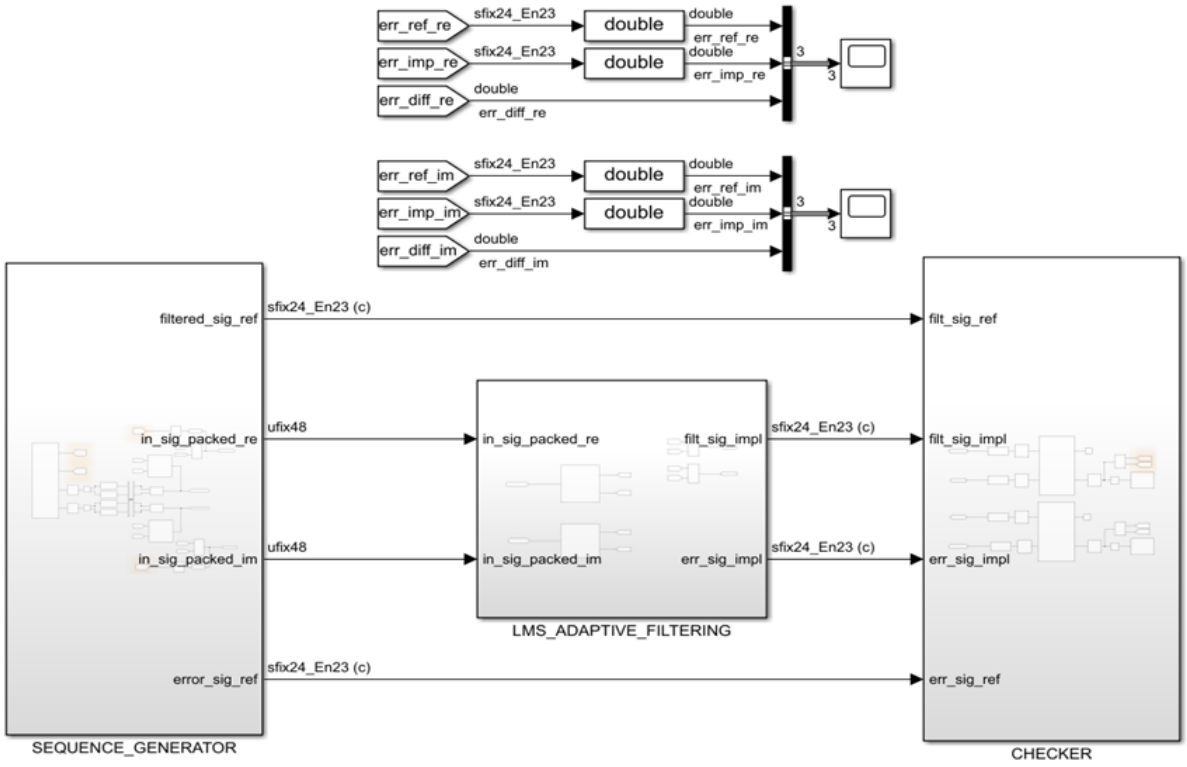

Figure 15. The top-level design of Simulink Reference Model

For a clear explanation, the sequence generator in figure 16 below is to generate the fixed-point inputs and the fixed-point expected outputs. The fixed-point inputs are driven into Simulink LMS Adaptive Filtering design to generate the corresponding implemented outputs which will be compared with the expected outputs. From the leftmost side, the pulse_generation function imports the Matlab testbench as mentioned previously in Section 5.1.1 to generate the floating-point desired input and noise input which are data sets of 1000 complex-valued samples going through Simulink Unbuffer block to form scalar sequences during fixed-step simulation before splitting into real path and imaginary path. Then, for each path, their inputs will be converted to the fixed-point data with Q-format 1.23 data type.

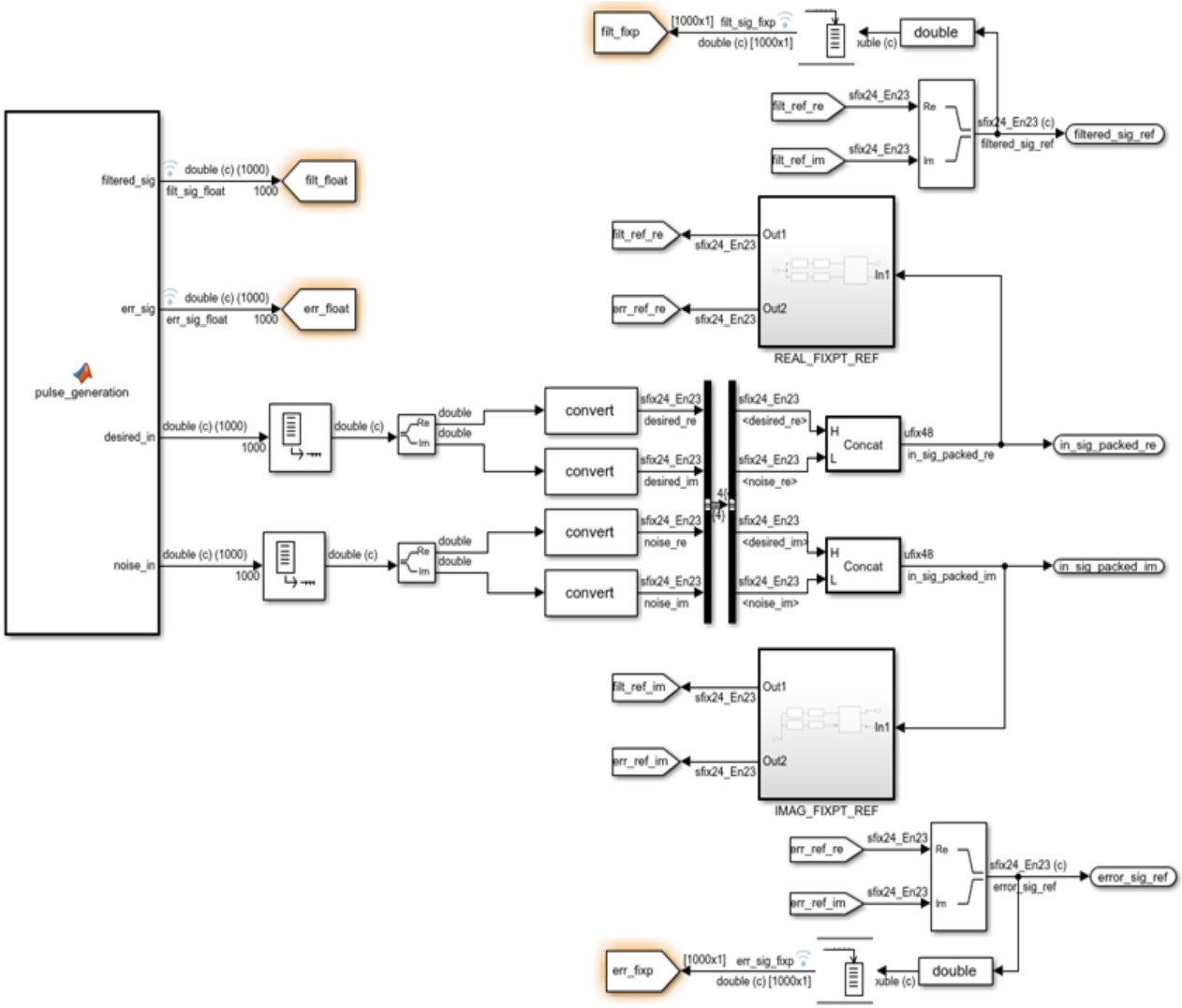

Figure 16. Sequence generator to generate fixed-point inputs and fixed-point expected outputs based on Matlab reference algorithm

To create the expected outputs to be compared with Simulink simulated outputs, these fixed-point inputs implement the fixed-point Matlab testbench specified in section 5.1.2 as imported function shown in figure 17 below. Concurrently, for each data path, their fixed-point inputs are cross connected to have a new datapack in which the first half represents the desired input whereas the last half represents the noise input.

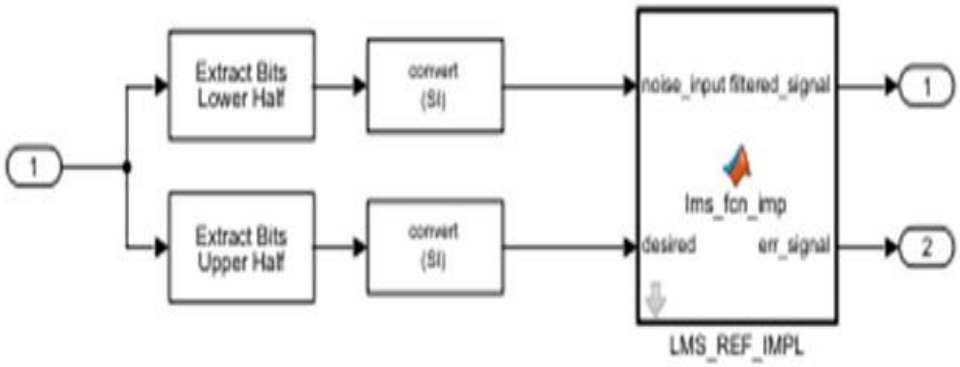

Figure 17. Generation of expected outputs by importing the fixed-point Matlab testbench

When it comes to the main adaptive LMS FIR filtering noise cancellation design, there are two 24-bits datasets extracted from the data packs derived from sequence generator. These indicated datasets, which are desired and noise reference, are the fixed-point inputs to implement the Matlab fixed-point algorithm. All of operations make the use of Simulink supported blocks to process left-to-right to abide by the dataflow as can be seen in figure 18. The only exception for not using Simulink block is the coefficient update procedure that uses the masked subsystem to call the Matlab fixed-point weight update logic function with specified design parameters, guarantees re-configurable data structure with modification of filter length variable. For more clear explanation, if the filter length parameter is changed to the large value for future innovation projects, the Simulink data model has to change significantly, especially creating more complicated cross connections in addition to large amounts of feedback paths for filter coefficient register, leads to the impossibility to create the functional model. Therefore, the available Matlab fixed-point weight update logic function is the optimal choice for weight update procedure.

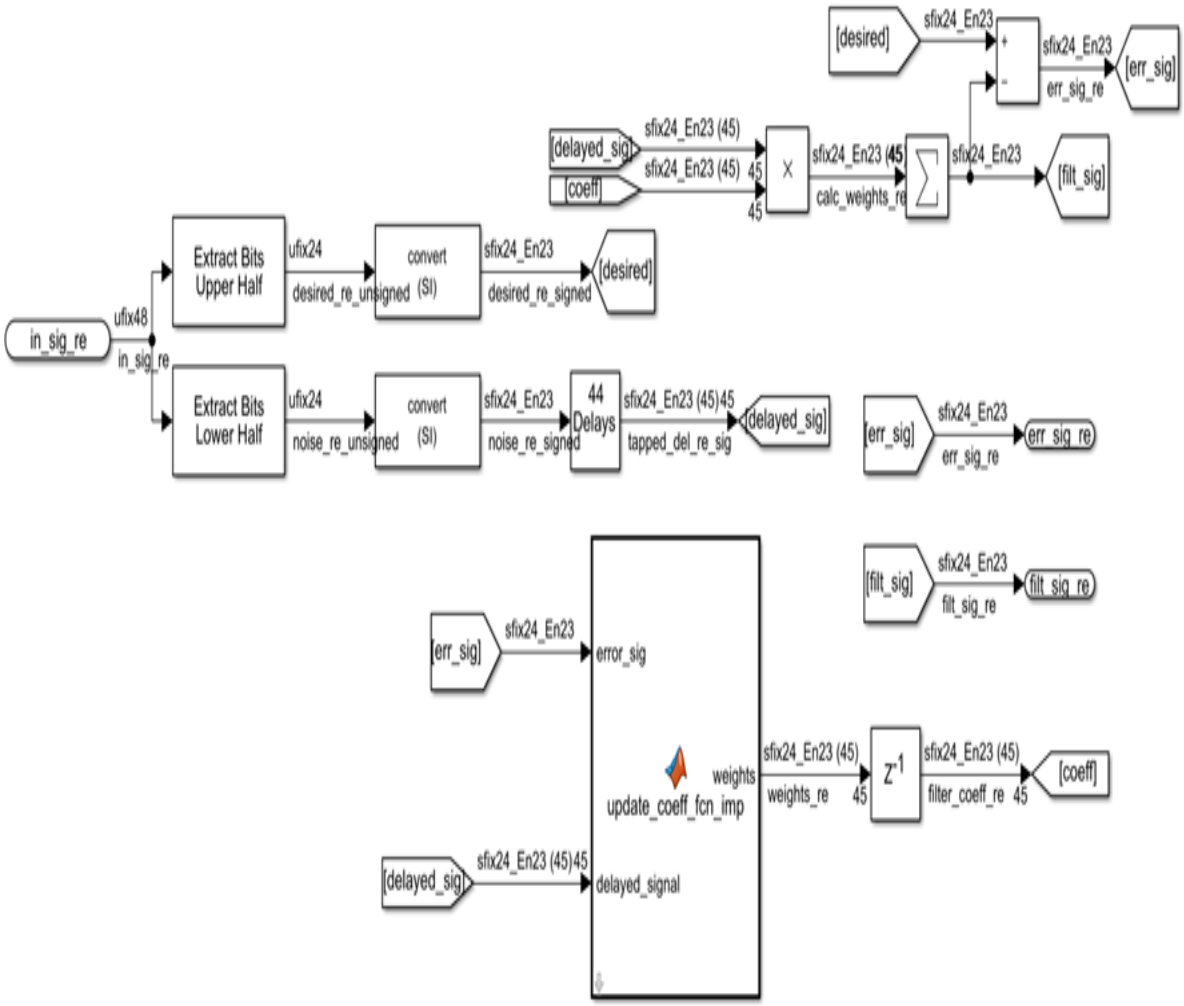

Figure 18. Simulink LMS design to implement adaptive noise cancellation process

After adaptive noise cancellation implementation, to verify model functional correctness, the checker shown in figure 19 verifies the equivalence checking between the golden reference output derived from fixed-point Matlab testbench implementation and the another one achieved from Simulink simulation.

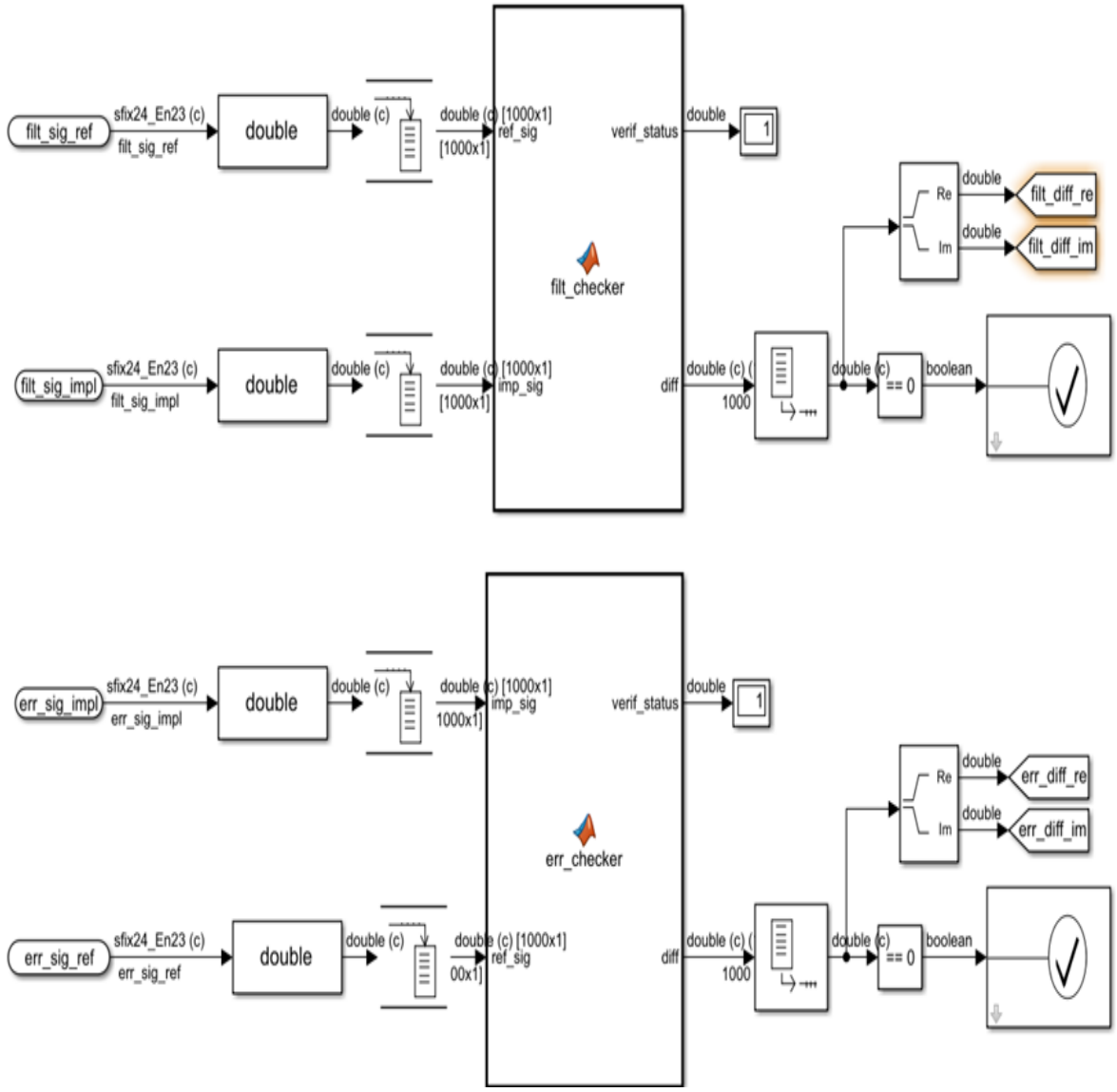

Figure 19. The checker to compare the implemented outputs with the expected outputs

This checker imports the Matlab lms_checker function demonstrated in listing 11 for the response checking of both filtered output and estimation error output. This function evaluates the size difference of the two inputs for dimensional comparability. The test result must be passed to move onto the next verification that compares the expected signal with the implemented signal for every corresponding element. The assertion messages will be displayed for compatible test cases during Simulink simulation.

```
function [verif_status, diff] = lms_checker(imp_sig, ref_sig )
ref_dim = size(ref_sig, 1);
imp_dim = size(imp_sig, 1);

if (ref_dim == imp_dim)

    dim_assert = true;

else
    dim_assert = false;
end


diff = complex(zeros(ref_dim,1),zeros(ref_dim,1));
mis_var = zeros(ref_dim,1);


if dim_assert

      disp(' Dimension checking is passed.')

      for i = 1:ref_dim
          exp = ref_sig(i);
          dut = imp_sig(i);
          diff(i) = abs(dut-exp);
          % Values do not match
          if ~isequal(dut,exp)
              disp(' !!! Data does not match.')
              verif_status = 0;
              % Something wrong. Find out which sample
              fprintf('  mismatch at index %5f, expected_ref: %0f,  dut_imp: %0f, diff: %9f, ratio: %5f \n', i, exp, dut, abs(dut-exp), dut/exp)
              error('Data mismatch happens.')
          else
              % The reference signal matches the implemented signal (design
              % under test)
              disp(' Data matches.');
              verif_status = 1;
          end

      end

else
    verif_status = 0;
    error('Signal dimension checking is failed');
end
```

Listing 11. Checker function to verify signal equivalence checking

The final step to verify is to run the Simulink reference model. For detailed waveform analysis of the output signals which can be observed in figure 20, the blue signal is the sinusoidal wave of estimation error signal whereas the red line indicates the error difference of zero which is calculated by subtracting the Simulink implemented data from the Matlab reference data.

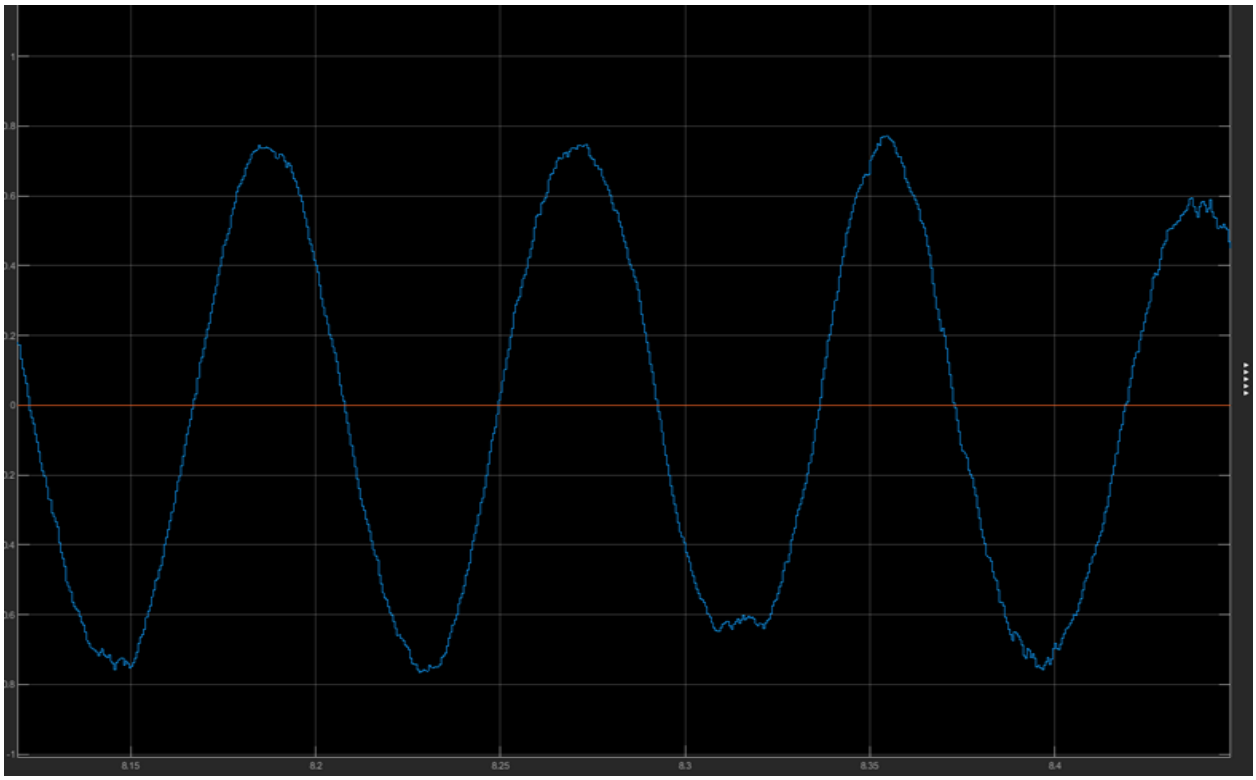

Figure 20. Graphical plot to show estimation error output and the zero difference between Matlab reference data and Simulink implemented data

In addition, the messages from lms_checker function for test-passed verification are on the Diagnostic Viewer display during Simulink simulation.

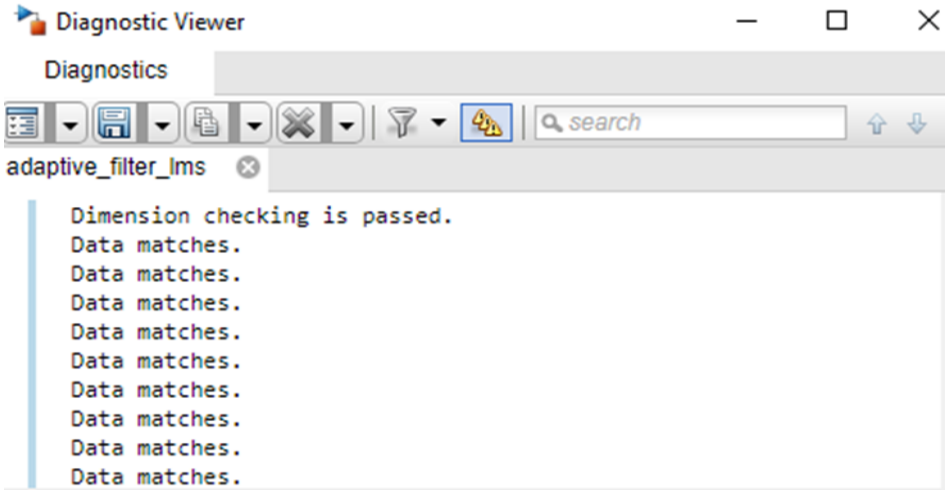

Figure 21. Diagnostic viewer of checker function during Simulink simulation

As figure 21 illustrated, the messages mean the Simulink reference model is designed as exactly specified as Matlab reference mathematical algorithm.

5.3 Register Transfer Level Design with VHDL

The final design phase is to create Register Transfer Level descriptions of the Simulink attributes. The RTL designs are required to have integrated HDL coding style to validate the Simulink reference model.

All RTL implementations are coded in VHDL according to design specifications and constraints mentioned in chapter 4. To be more specific, fixed predefined design parameters are configured in LMS_ADAPTIVE_FILTERING_pkg.vhd file as illustrated in listing 12. Apart from initial declarations of fixed parameters such as filter coefficient length and weight update control logics, the constant w_data_c of 24 is the common data word length configuration for all data signals which are design entity in-puts, entity outputs and the outputs of different binary arithmetic operations. Besides, w_mult_c of 48 and w_adder_c of 30 are the constants used for multiplication and finite convolution sum. Furthermore, data word length of step size is to obey the specified design specification that its fractional length is 30 and its word length is 24. In addition, it is fixed to be an unsigned binary of 0.008 which hexadecimal converted result is 83126F.

```
PACKAGE LMS_ADAPTIVE_FILTERING_pkg IS
  constant w_data_c  : natural   := 1 + 0 + 23;
  constant w_mult_c  : natural   := w_data_c * 2;
  constant coeff_len : natural   := 45;
  constant msb_c     : natural   := coeff_len-1;
  constant log_coeff_len  : natural   := 6;
  constant w_adder_c      : natural   := w_data_c + log_coeff_len ;
  constant reset_weights  : std_logic   := '0';
  constant update_weights : std_logic   := '1';
  constant step_size_c    : unsigned(w_data_c-1 DOWNTO 0)   := to_unsigned(16#83126F#, w_data_c);
  constant step_s_fracw   : natural   := 30;


  TYPE vector_of_signed_w_data IS ARRAY (NATURAL RANGE <>) OF signed(w_data_c-1 DOWNTO 0);
  TYPE vector_of_signed_w_mult IS ARRAY (NATURAL RANGE <>) OF signed(w_mult_c-1 DOWNTO 0);
```

Listing 12. Data declarations of fixed parameters in HDL package file

As specified in section 4.2.2 about overflow handling and rounding, round and wrap on overflow is the chosen method to be coded as an HDL procedure that can be seen in listing 13. Its inputs are based on the following factors: rounding Most Significant Bit, rounding Least Significant Bit and wrapping Most Significant Bit. There are two concurrent procedures that the rounding procedure uses resize function to truncate input to

data with a new specified data width but retain the sign bit as can be seen in line 75 whereas the wrapping procedure is going through recursive OR operations with the usage of for loop as coded from line 80 to line 82 before using different bitwise calculations based on the wrapping theory to generate the wrap bit. The result from rounding and wrapping procedures is the addition of the rounded variable and the wrap bit. More-over, the cumulative sum of elements procedure is also directly coded at the top of the identical HDL package file as can be seen from line 45 to line 61.

```
44  PACKAGE BODY LMS_ADAPTIVE_FILTERING_pkg is
45  procedure cusum (
46      constant add_len          : in natural;
47      constant expected_data_w  : in natural;
48      signal in_sig             : in vector_of_signed_w_data(0 to msb_c);
49      signal out_sig            : out signed(w_data_c-1 DOWNTO 0)) is
50
51      variable inter_sig_v      : signed(add_len-1 DOWNTO 0);
52  begin
53      inter_sig_v := to_signed(16#000000#, add_len);
54
55      FOR i IN 0 TO msb_c LOOP
56        inter_sig_v := inter_sig_v + resize(in_sig(i),add_len);
57      END LOOP;
58
59      out_sig <= inter_sig_v(w_data_c-1 DOWNTO 0);
60
61  end procedure cusum;
62
63  procedure round_and_wrap (
64      constant rounding_msb     : in natural;
65      constant rounding_lsb     : in natural;
66      constant wrapping_msb     : in natural;
67      signal in_sig             : in signed(w_mult_c-1 DOWNTO 0);
68      signal out_sig            : out signed(w_data_c-1 DOWNTO 0)) is
69
70      variable inter_a_v        : std_logic;
71      variable inter_b_v        : signed(1 downto 0);
72      variable inter_sig_v      : signed(w_data_c-1 DOWNTO 0);
73  begin
74      --rounding procedure
75      inter_sig_v := resize(in_sig(rounding_msb downto rounding_lsb),w_data_c);
76
77      --wrapping procedure
78      inter_a_v := in_sig(0);
79
80      FOR i IN 1 TO rounding_lsb-2 LOOP
81        inter_a_v := inter_a_v OR in_sig(i);
82      END LOOP;
83
84      inter_b_v := ('0' & (in_sig(rounding_lsb-1) AND (( NOT in_sig(wrapping_msb)) OR(inter_a_v))));
85
86      out_sig <= inter_sig_v + inter_b_v;
87
88  end procedure round_and_wrap;
```

Listing 13. HDL procedures for overflow handling and cumulative sum computation

The main RTL coding for adaptive noise cancellation algorithm is separated into three partial operations which run simultaneously. RTL coding style uses rising clock edge to drive the signal logic and asynchronous reset which will be activated when the reset signal is asserted. To be more explicit, tapped delay processing described in listing 14 is to generate 45-taps noise reference input.

```
Tapped_Delay_process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    Tapped_Delay_reg <= (OTHERS => to_signed(16#000000#, w_data_c));
  ELSIF clk'EVENT AND clk = '1' THEN
    IF valid_in = '1' THEN
      Tapped_Delay_reg(msb_c-1) <= noise_re_signed;
      Tapped_Delay_reg(0 TO msb_c-2) <= Tapped_Delay_reg(1 TO msb_c-1);
    END IF;
  END IF;
END PROCESS Tapped_Delay_process;

tapped_del_re_sig(0 TO msb_c-1) <= Tapped_Delay_reg(0 TO msb_c-1);
tapped_del_re_sig(msb_c) <= noise_re_signed;
```

Listing 14. HDL procedure to create tapped delay signal from noise input

In addition, as can be seen in listing 15, adaptive weight update from line 90 to line 99 is to utilize mathematical computations based on adaptive LMS FIR filtering theory. For every clock cycle, valid signal starts filter coefficient registering.

```
90  step_sig_first <= step_size_signed * err_sig_re_signed;
91  step_sig_cst <= step_sig_first(w_mult_c-1 DOWNTO 0);
92  round_and_wrap(step_sig_cst'length-1,step_s_fracw,step_sig_cst'length-1,step_sig_cst,step_sig);
93
94  weights_re_gen: FOR i IN 0 TO msb_c GENERATE
95    adapt_corr_cst(i) <= tapped_del_re_sig(i) * step_sig;
96    round_and_wrap(adapt_corr_cst(i)'length-2,w_data_c-1,adapt_corr_cst(i)'length-1,adapt_corr_cst(i),adaptive_correction(i));
97    updated_weight(i) <= adaptive_correction(i) + filter_coeff_re_signed(i);
98    weights_re(i) <= updated_weight(i);
99  END GENERATE weights_re_gen;
100 --
101 Delay_process : PROCESS (clk, reset)
102 BEGIN
103   IF reset = '1' THEN
104     filter_coeff_re <= (OTHERS => to_signed(16#000000#, w_data_c));
105   ELSIF clk'EVENT AND clk = '1' THEN
106     IF valid_in = '1' THEN
107       filter_coeff_re <= weights_re;
108     END IF;
109   END IF;
110 END PROCESS Delay_process;
111
112 calc_weights_re_gen: FOR k IN 0 TO msb_c GENERATE
113   calc_weights_re_cst(k) <= tapped_del_re_sig(k) * filter_coeff_re_signed(k);
114   round_and_wrap(calc_weights_re_cst(k)'length-2,w_data_c-1,calc_weights_re_cst(k)'length-1,calc_weights_re_cst(k),calc_weights_re(k));
115 END GENERATE calc_weights_re_gen;
116 --
117 cusum(w_adder_c,w_data_c,calc_weights_re,filt_sig_re_signed);
118 --
```

Listing 15. RTL top-down adaptation process

As coded from line 112 to line 117 in listing 15, FIR filtering of the delayed noise input according to the updated filter coefficient set and the tap delay of noise reference is the vector-vector multiplication to create a new set of filter weights which then goes through the rounding and wrapping procedure. After that, the adaptive filtered output is generated by using cusum procedure declared in HDL package file for finite summation of the calculated filter weight.

6 Adaptive FIR Filtering Noise Cancellation Module Verification

The verification techniques used to verify RTL design of Adaptive FIR Filtering Noise Cancellation module are explained here and the results of these verifications are also presented. RTL design in this Thesis goes through multiple levels of verification. QuetaSim as an HDL Simulator empirically checks simulation-based RTL functional characteristics with assertions and provides the design code coverage analysis. Besides, HDL Cosimulation establishes mutual connections between Simulink and QuestaSim for parallel simulations to guarantee that RTL design does what Simulink reference modeling expects under clock synchronization condition. Finally, an UVM based bit-exact simulation testbench is used to verify the specific use case. The HDL entities are integrated to the UVM environment. The DPI-C mathematical model, which is converted from the Simulink reference model, is to run to generate input stimulus and the reference output data. During UVM test simulation, the generated input stimulus will stimulate RTL entities then their output data will be captured to be compared with the DPI-C generated reference output data.

6.1 HDL Testbench Generation and Simulation with Questa Sim

This first verification to verify RTL design is to generate a HDL test bench to simulate HDL design files in QuestaSim simulator.

This test bench verifies the created HDL DUT against test vectors generated from Simulink reference model. The input stimulus and expected data, which are generated from Simulink simulation, are stored in data files (.dat). In detail, the input data are in_sig_packed_re, in_sig_packed_im and the output data are err_sig_impl_re_expected, err_sig_impl_im_expected, filt_sig_impl_re_expected, filt_sig_impl_im_expected that their data format is hexadecimal number. During HDL simulation, the HDL test bench will read the input stimulus derived from the input data files then drive these stimuli to DUT input ports as illustrated in listing 16. After that, the HDL testbench captures the actual DUT outputs which will be compared with the corresponding expected outputs stored in the output data files.

```
u_LMS_ADAPTIVE_FILTERING : LMS_ADAPTIVE_FILTERING
  PORT MAP( clk => clk,
            reset => reset,
            valid_in => valid_in,
            in_sig_packed_re => in_sig_packed_re_out,   -- ufix48
            in_sig_packed_im => in_sig_packed_im_out,   -- ufix48
            valid_out => valid_out,
            filt_sig_impl_re => filt_sig_impl_re,   -- sfix24_En23
            filt_sig_impl_im => filt_sig_impl_im,   -- sfix24_En23
            err_sig_impl_re => err_sig_impl_re,   -- sfix24_En23
            err_sig_impl_im => err_sig_impl_im   -- sfix24_En23
            );
```

Listing 16. RTL entities binding to the HDL testbench

As can be seen in listing 17, for real data path, the process statement describes the procedure which opens in_sig_packed_re.dat file to read the input stimulus as read_data variable then stores it as a raw data for the registering process. Because stimulus capture is dependent on not only system clock signal but also intrinsic rdEnb variable value, there are two concurrent processes that the first one as coded from line 259 to line 266 is responsible to hold raw data at every rising clock edge to be driven into offset data if sequence reading is disabled, whereas the remaining one as coded from line 268 to line 275 is to immediately achieve raw data as an offset data if there occurs any change of raw data.

```
243   -- Data source for in_sig_packed_re
244   in_sig_packed_re_fileread: PROCESS (async_time_del, tb_enb_delay, rdEnb)
245     FILE fp: TEXT open READ_MODE is "in_sig_packed_re.dat";
246     VARIABLE l: LINE;
247     VARIABLE read_data: std_logic_vector(47 DOWNTO 0);
248
249   BEGIN
250     IF tb_enb_delay /= '1' THEN
251     ELSIF rdEnb = '1' AND NOT ENDFILE(fp) THEN
252       READLINE(fp, l);
253       HREAD(l, read_data);
254     END IF;
255     rawData_in_sig_packed_re <= unsigned(read_data(47 DOWNTO 0));
256   END PROCESS in_sig_packed_re_fileread;
257
258   -- holdData reg for sequence_generator
259   stimuli_sequence_generator_process: PROCESS (clk, reset)
260   BEGIN
261     IF reset = '1' THEN
262       holdData_in_sig_packed_re <= (OTHERS => 'X');
263     ELSIF clk'event AND clk = '1' THEN
264       holdData_in_sig_packed_re <= rawData_in_sig_packed_re;
265     END IF;
266   END PROCESS stimuli_sequence_generator_process;
267
268   stimuli_sequence_generator: PROCESS (rawData_in_sig_packed_re, rdEnb)
269   BEGIN
270     IF rdEnb = '0' THEN
271       in_sig_packed_re_offset <= holdData_in_sig_packed_re;
272     ELSE
273       in_sig_packed_re_offset <= rawData_in_sig_packed_re;
274     END IF;
275   END PROCESS stimuli_sequence_generator;
276
277   in_sig_packed_re <= in_sig_packed_re_offset AFTER 0 ns;
```

Listing 17. Sequence generation to drive stimulus to the HDL Design Under Test

The HDL testbench is configured that it opens the reference data .dat file then reads the captured data as expected outputs. After that, the expected outputs will be compared with RTL implemented outputs. As a sample demonstration, the verification of real-valued filtered output is shown in listing 18. Expected filtered output reading procedure is coded from line 374 to line 388 and the equivalence checking process is coded from line 390 to line 402 in which at every valid rising clock edge, there is an assertion to verify whether the actual RTL output is equal to the expected output or not.

```
372  filt_sig_impl_re_signed <= signed(filt_sig_impl_re);
373  -- Data source for filt_sig_impl_re_expected
374  filt_sig_impl_re_expected_fileread: PROCESS (filt_sig_impl_re_addr_delay, tb_enb_delay, valid_out)
375    FILE fp: TEXT open READ_MODE is "filt_sig_impl_re_expected.dat";
376    VARIABLE l: LINE;
377    VARIABLE read_data: std_logic_vector(23 DOWNTO 0);
378
379  BEGIN
380    IF tb_enb_delay /= '1' THEN
381    ELSIF valid_out = '1' AND NOT ENDFILE(fp) THEN
382      READLINE(fp, l);
383      HREAD(l, read_data);
384    END IF;
385    filt_sig_impl_re_expected <= signed(read_data(23 DOWNTO 0));
386  END PROCESS filt_sig_impl_re_expected_fileread;
387
388  filt_sig_impl_re_ref <= filt_sig_impl_re_expected;
389
390  filt_sig_impl_re_signed_checker: PROCESS (clk, reset)
391  BEGIN
392    IF reset = '1' THEN
393      filt_sig_impl_re_testFailure <= '0';
394    ELSIF clk'event AND clk = '1' THEN
395      IF valid_out = '1' AND filt_sig_impl_re_signed /= filt_sig_impl_re_ref THEN
396        filt_sig_impl_re_testFailure <= '1';
397        ASSERT FALSE
398          REPORT "Error in filt_sig_impl_re_signed: Expected " & to_hex(filt_sig_impl_re_ref) & (" Actual " & to_hex(filt_sig_impl_re_signed))
399          SEVERITY ERROR;
400      END IF;
401    END IF;
402  END PROCESS filt_sig_impl_re_signed_checker;
```

Listing 18. HDL equivalence checking of actual RTL output and Simulink expected output

After HDL compilation, QuestaSim is used to run the configure HDL testbench to verify the functional correctness of RTL design. The detailed waveform shown in figure 22 is that the test result is passed after a completed simulation.

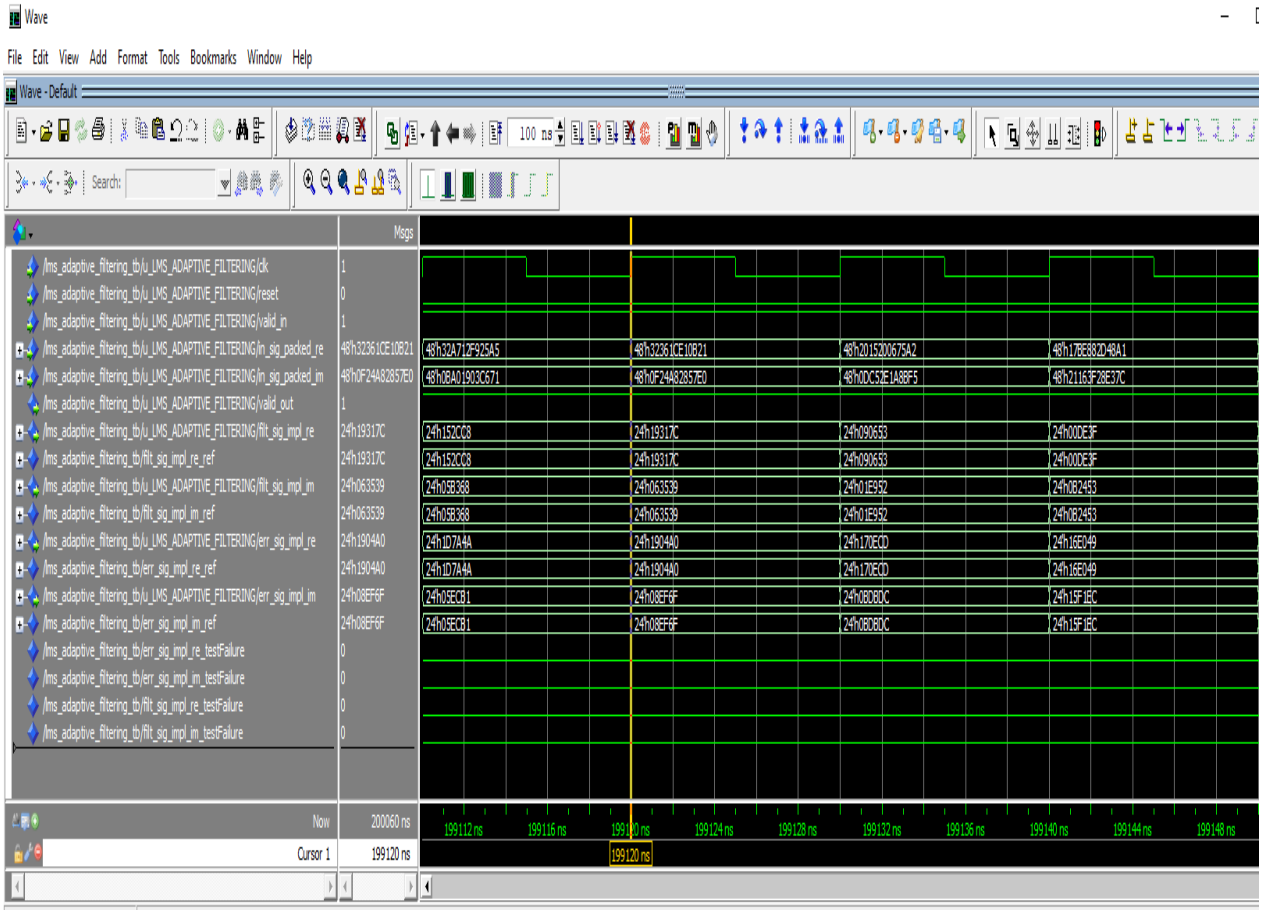

Figure 22. Detailed waveform of signals from HDL testbench verification by QuestaSim

For an instance explanation, observing at 199120 ns, all RTL implemented outputs are equivalence to the respective reference outputs and their asserted variables for test failure checking are always zero during test simulation.

6.2 Code Coverage

Code coverage is a technique to collect the statistics on the execution of each line of HDL code. Its coverage analysis is evaluated using the built-in features of QuestaSim to measures how many times certain aspects of the source are exercised while running a suite of tests. Code Coverage types are divided into Statements, Branches, FEC Expressions, FEC Conditions, Finite State Machine and Toggles. [16,41-48].

Coverage Summary By Instance:

Scope	TOTAL	Statement	Branch	Toggle
TOTAL	100.00	100.00	100.00	100.00
u_LMS_ADAPTIVE_FILTERING	100.00	100.00	--	100.00
u_LMS_ADAPT_REAL	100.00	100.00	100.00	100.00
u_LMS_ADAPT_IMAG	100.00	100.00	100.00	100.00

Local Instance Coverage Details:

Total Coverage:					100.00%	100.00%
Coverage Type	**Bins**	**Hits**	**Misses**	**Weight**	**% Hit**	**Coverage**
Statements	5	5	0	1	100.00%	100.00%
Toggles	582	582	0	1	100.00%	100.00%

Recursive Hierarchical Coverage Details:

Total Coverage:					100.00%	100.00%
Coverage Type	**Bins**	**Hits**	**Misses**	**Weight**	**% Hit**	**Coverage**
Statements	591	591	0	1	100.00%	100.00%
Branches	16	16	0	1	100.00%	100.00%
Toggles	1834	1834	0	1	100.00%	100.00%

Figure 23. Code coverage result of the main HDL coding of the design module

For RTL design of Adaptive LMS FIR Filtering Noise Cancellation module, there are only Statements, Branches and Toggle coverages needed for statistics analysis. The achieved results are shown in figure 23 and figure 24.

Coverage Summary By Instance:

Scope	TOTAL	Statement	FEC Expression
TOTAL	100.00	100.00	100.00
lms_adaptive_filtering_pkg	100.00	100.00	100.00
cusum	100.00	100.00	--
round_and_wrap	100.00	100.00	100.00

Local Instance Coverage Details:

Total Coverage:					100.00%	100.00%
Coverage Type	**Bins**	**Hits**	**Misses**	**Weight**	**% Hit**	**Coverage**
Statements	14	14	0	1	100.00%	100.00%
FEC Expressions	2	2	0	1	100.00%	100.00%

Recursive Hierarchical Coverage Details:

Total Coverage:					100.00%	100.00%
Coverage Type	**Bins**	**Hits**	**Misses**	**Weight**	**% Hit**	**Coverage**
Statements	14	14	0	1	100.00%	100.00%
FEC Expressions	2	2	0	1	100.00%	100.00%

Figure 24. Code coverage result of HDL coding of design package file

For both real path and imaginary path, code coverage reached 100% as the desired goal mentioned in section 4.2.4. Therefore, the verification for code coverage of RTL design was considered done.

6.3 HDL Cosimulation Verification

Co-simulation automatically drives stimulus into an HDL model from Simulink testbench and runs a RTL simulator concurrently. Cosimulation compares expected output derived from Simulink testbench to the HDL model's output. HDL Verifier is required to be installed. Co-simulation compares the models bit-accurately and cycle-accurately. [12,32.] Simulink configuration is presented in figure 25 below.

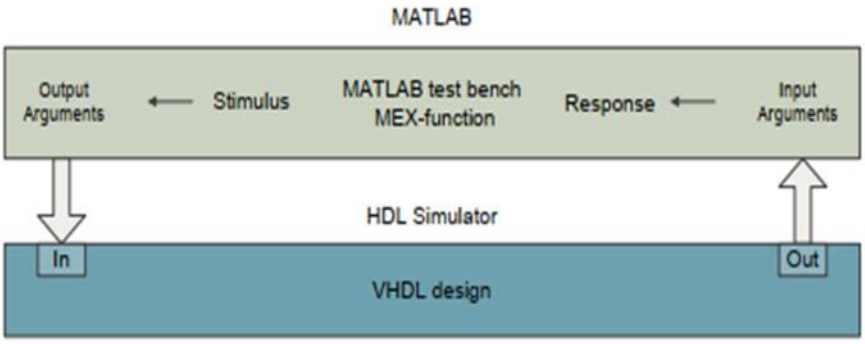

Figure 25. HDL Co-simulation configuration (Reprinted from [12,32])

Mathworks reference documentation [23] briefly described the procedure of how to set up a communication link between Simulink and HDL simulator through HDL Cosimulation, stating that:

> In co-simulation, it creates a communication link between the HDL simulator and Simulink. Such a link enables to verify a model directly against the HDL implementation, create test signals and testbench for HDL code, use a behavioral model as a reference in an HDL simulation, use analysis and visualization features for real-time insight into an HDL implementation, integrate a new model with the existing HDL design. [23.]

Error is calculated from the differences of Simulink model simulation and RTL simulation. Start time alignment is configured for delay balancing between Simulink and RTL simulator. If RTL is written correctly to perform functional characteristics of Simulink model, then the error should be zero. The comparison is done on the output ports and it is a rapid method to verify HDL design correctness every time the reference algorithm is adjusted. [12,32.]

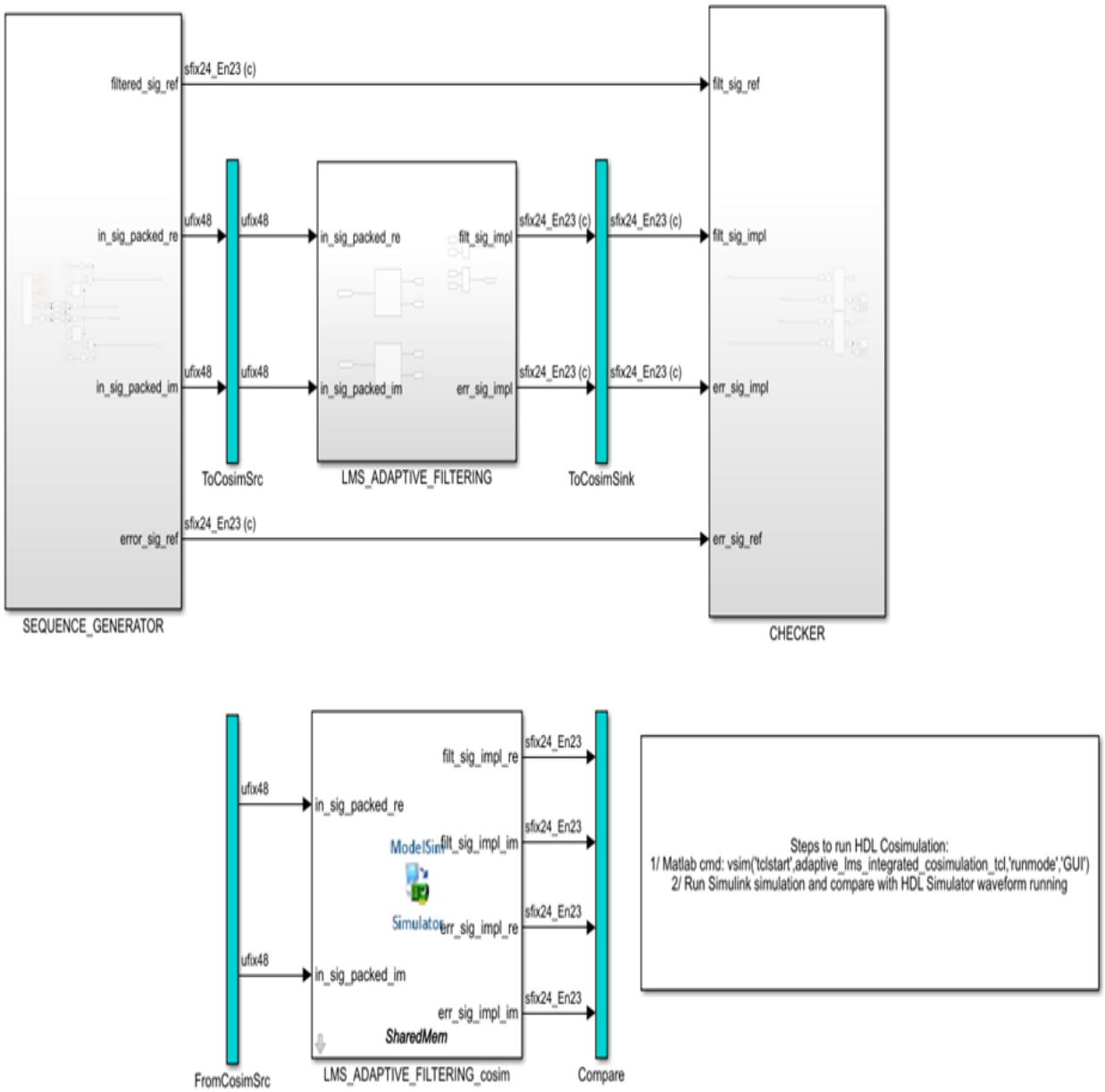

Figure 26. Simulink cosimulation model of adaptive FIR filtering noise cancellation

To be more specific, the Simulink reference model that includes an HDL Cosimulation block shown in figure 26 above will run the generated HDL code in QuestaSim HDL simulator. This block drives Simulink stimulus into RTL input ports of DUT then captures the actual outputs from DUT to compare against Simulink reference outputs of LMS_ADAPTIVE_FILTERING subsystem.

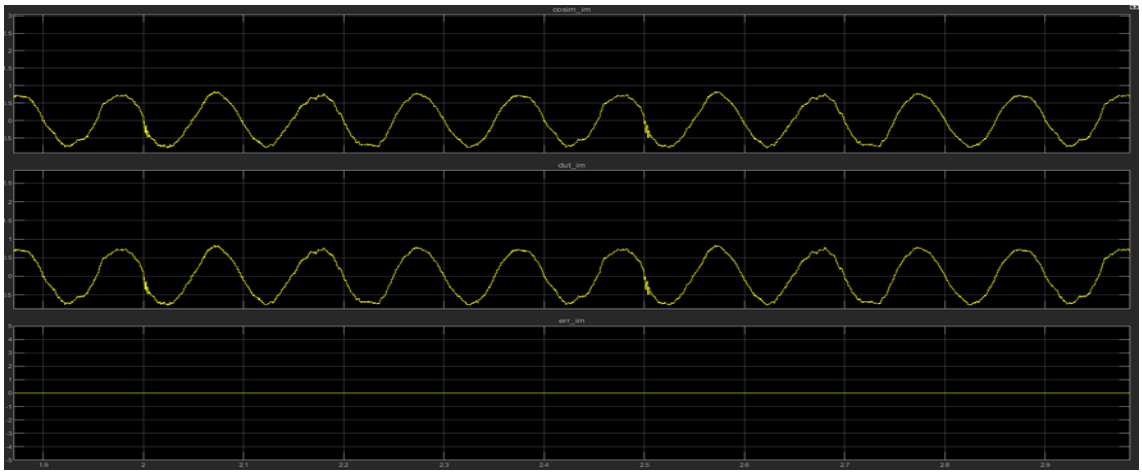

Figure 27. The comparison plot to show Simulink estimation error output and its RTL output

The plot presented in figure 27 is to prove the correctness of RTL design against Simulink mathematical model. The topmost sinusoidal wave is the achieved estimation error output derived from DUT meanwhile the middle one represents its Simulink simulated output. As can be seen, these sinusoidal waves have the identical characteristics. The bottom line is the error difference being zero over time simulation between DUT output and Simulink reference output. Therefore, the design functional characteristics are correctly converted from Simulink modeling to RTL design coding.

6.4 Functional Verification with Universal Verification Methodology

For this Thesis project, Direct Programming Interface was utilized to create a UVM test environment. This environment contains a System Verilog DPI top module which is binding to RTL design under verification. The UVM test bench basically includes a sequence, a scoreboard, and design under verification. As demonstrated in figure 28, the top module which comprises clock and reset signals to control system process, instantiates Design Under Verification (DUV) which is connected to verification environment through module interface. The UVM environment consists of the scoreboard and the agent which includes a sequencer, a monitor and a driver. The detailed process of UVM testbench is that the sequence object which defines a set of transactions, is divided into two paths. For the first path, the sequencer is responsible for routing the stimulus data from the sequence to the driver which will transform each transaction to the desired protocol and drives the transaction to the DUV through the virtual interface. After RTL operations of DUV, the passive UVM monitor samples implemented output data to send to the scoreboard. The remaining path is a direct path in which the scoreboard receives the reference

output data from the sequence object then compares this expected data with the implement data from RTL DUV.

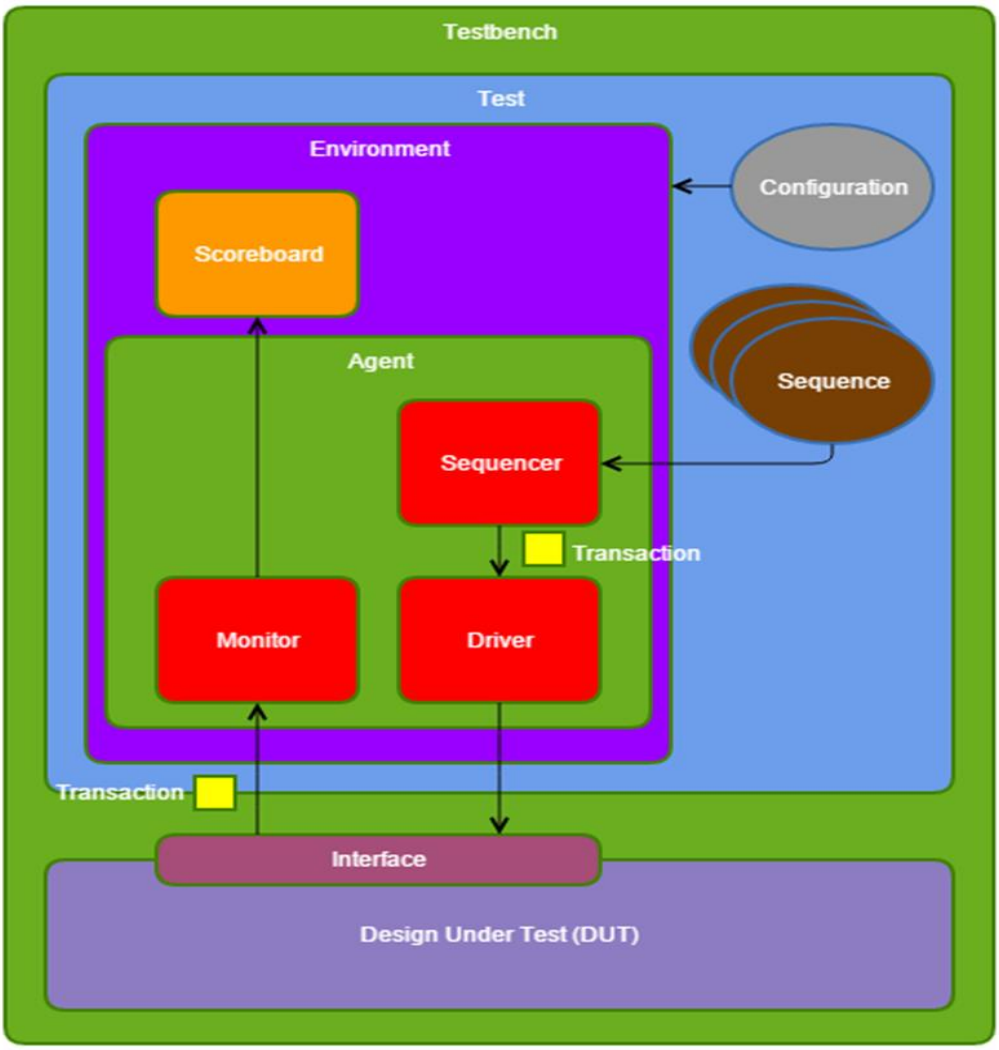

Figure 28. UVM Testbench Architecture (Reprinted from [25])

To be more precise, Simulink subsystems are exported as generated C codes which describe design implementations to be integrated with UVM environment with a direct programming interface (DPI). It means that UVM sequence, DUV, and UVM scoreboard are generated from Sequence Generator, LMS Adaptive Filtering design and Checker subsystem respectively. These DPI-C codes can be found at subsystem build directories that there are subsystem DPI package files for function declarations, the generated System Verilog DPI subsystem, and the DPI-C subsystem wrappers for C descriptions of Simulink design operations.

UVM hierarchy configuration is briefly demonstrated in lms_pkg.sv file as can be seen in figure 29.

```
`include "uvm_macros.svh"
package lms_pkg;
import uvm_pkg::*;
//Sequence item, Sequence and Sequencer
`include "../sequence/mw_LMS_FILTERING_sequence_trans.sv"
`include "../sequence/mw_LMS_FILTERING_sequence.sv"
//DUT transaction object
`include "../uvm_artifacts/mw_LMS_FILTERING_trans.sv"
//Driver, monitor and agent
`include "../uvm_artifacts/mw_LMS_FILTERING_driver.sv"
`include "../uvm_artifacts/mw_LMS_FILTERING_monitor.sv"
`include "../uvm_artifacts/mw_LMS_FILTERING_monitor_input.sv"
`include "../uvm_artifacts/mw_LMS_FILTERING_agent.sv"
//Scoreboard
`include "../scoreboard/mw_LMS_FILTERING_scoreboard.sv"
//UVM environment and test
`include "../uvm_artifacts/mw_LMS_FILTERING_environment.sv"
`include "../uvm_artifacts/mw_LMS_FILTERING_test.sv"
endpackage: lms_pkg
```

Figure 29. UVM Hierarchy as presented in HDL package file

The functionalities of UVM artifacts as presented in figure 29 are defined below:

• mw_LMS_FILTERING_sequence_trans.sv – This file includes type definitions of sequence items

• mw_LMS_FILTERING_sequence.sv – This file contains sequence declarations and sequencer instantiation.

• mw_LMS_FILTERING_trans.sv - This file contains a UVM object that defines the input transaction type for the scoreboard.

• mw_LMS_FILTERING_driver.sv - This file includes a pass-through UVM driver by default.

• mw_LMS_FILTERING_monitor.sv - This file includes a UVM monitor. The monitor samples implemented signals from the DUV to the scoreboard.

• mw_LMS_FILTERING_monitor_input.sv - This file includes a pass-through UVM monitor. The monitor samples expected signals from the driver to the scoreboard.

• mw_LMS_FILTERING_agent.sv - This file includes a UVM agent that instantiates sequence, driver, and monitor.

• mw_LMS_FILTERING_scoreboard.sv – This file consists of a UVM scoreboard

• mw_LMS_FILTERING_environment.sv - This file includes a UVM environment, that instantiates an agent and a scoreboard.

• mw_LMS_FILTERING_test.sv - This file includes a UVM test which is based on the specified use case of adaptive LMS filtering design.

After completing UVM configurations, the System Verilog top design illustrated in listing 19 is integrated with UVM environment. The UVM interface as coded in mw_LMS_FILTERING_if.sv file is to connect with DPI-C representation of Simulink LMS Adaptive Filtering design subsystem. During UVM simulation, it receives stimulus from UVM driver then drives them into the scoreboard through UVM monitor for UVM response checking. In addition, the top design utilizes this interface to capture DPI-C implemented output signals to be compared with RTL outputs which is delayed for signal compatibility. The RTL binding can be seen from line 43 to line 53.

```
41  mw_LMS_FILTERING_if dutif ();
42
43  LMS_ADAPTIVE_FILTERING u_LMS_ADAPTIVE_FILTERING (.in_sig_packed_re(in_sig_packed_re),  /* ufix48 */
44                                                   .in_sig_packed_im(in_sig_packed_im),  /* ufix48 */
45                                                   .clk(dutif.clk),
46                                                   .reset(reset),
47                                                   .valid_in(dutEnable),
48                                                   .filt_sig_impl_re(filt_sig_impl_re),  /* sfix24_En23 */
49                                                   .filt_sig_impl_im(filt_sig_impl_im),  /* sfix24_En23 */
50                                                   .err_sig_impl_re(err_sig_impl_re),  /* sfix24_En23 */
51                                                   .err_sig_impl_im(err_sig_impl_im),  /* sfix24_En23 */
52                                                   .valid_out(valid_out)
53                                                   );
54
55  LMS_FILTERING_dpi dut(.clk(dutif.clk),.clk_enable(dutif.clk_enable),.reset (dutif.rst),.in_sig_packed_re (dutif.in_sig_packed_re),
56                        .in_sig_packed_im (dutif.in_sig_packed_im),.Out1_re (dutif.Out1_re),.Out1_im (dutif.Out1_im),
57                        .Out3_re (dutif.Out3_re),.Out3_im (dutif.Out3_im));
```

Listing 19. Instantiations of UVM interface, RTL DUV, and Simulink DPI-C representation

The equivalence checking between real-valued RTL implemented estimation error and its DPI-C representation is demonstrated in listing 20. The signal err_sig_impl_re_delayed is the delayed RTL estimation error output whereas err_sig_impl_re_ref is its System Verilog DPI-C generation. The equivalence checking is passed, leads to that err_sig_impl_re_testFailure value is always zero.

```
always
  begin : err_sig_impl_re_checker
    # (5);
    if (reset == 1'b1) begin
      err_sig_impl_re_testFailure <= 1'b0;
    end
    else begin
      err_sig_impl_re_testFailure <= 1'b0;
      if (dutEnable == 1'b1)

          if(err_sig_impl_re_delayed !== err_sig_impl_re_ref) begin
              err_sig_impl_re_testFailure <= 1'b1;
              $display("ERROR in err_sig_impl_re at time %t : Expected '%h' Actual '%h'", $time, err_sig_impl_re_ref, err_sig_impl_re_delayed );
          end
      end
    end

assign err_sig_impl_im_ref = dutif.Out3_im[23:0];

always
  begin : err_sig_impl_im_checker
    # (5);
    if (reset == 1'b1) begin
      err_sig_impl_im_testFailure <= 1'b0;
    end
    else begin
      err_sig_impl_im_testFailure <= 1'b0;
      if (dutEnable == 1'b1)

          if(err_sig_impl_im_delayed !== err_sig_impl_im_ref) begin
              err_sig_impl_im_testFailure <= 1'b1;
              $display("ERROR in err_sig_impl_im at time %t : Expected '%h' Actual '%h'", $time, err_sig_impl_im_ref, err_sig_impl_im_delayed );
          end
      end
    end


assign isTestFailed = err_sig_impl_im_testFailure | (err_sig_impl_re_testFailure | (filt_sig_impl_re_testFailure | filt_sig_impl_im_testFailure));



always
  begin : completed_msg
    # (5);
    if (testDone == 1'b1) begin
      if (isTestFailed == 1'b0) begin
        $display("***************TEST COMPLETED (PASSED)***************");
      end
      else begin
        $display("***************TEST COMPLETED (FAILED)***************");
      end
    end
  end
```

Listing 20. Top-module equivalence checking in UVM environment written by System Verilog

The same coding technique is applied for filtered output at both real path and imaginary path. If all of these checking are done without failure during UVM simulation then the test-completed message is displayed, otherwise the test-failed message is shown instead.

```
# UVM_INFO verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(215) @ 0: reporter [Questa UVM] QUESTA_UVM-1.2.3
# UVM_INFO verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(217) @ 0: reporter [Questa UVM]  questa_uvm::init(+struct)
# UVM_INFO @ 0: reporter [RNTST] Running test mw_LMS_FILTERING_test...
# **************TEST COMPLETED (PASSED)**************
# **************TEST COMPLETED (PASSED)**************
# **************TEST COMPLETED (PASSED)**************
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1268) @ 200030: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :    4
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [Questa UVM]     2
# [RNTST]     1
# [TEST_DONE]     1
# ** Note: $finish    : C:/questasim64_10.4e/win64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
#    Time: 200030 ns  Iteration: 78  Instance: /mw_LMS_FILTERING_top
```

Figure 30. UVM testbench simulation with test-passed result

After HDL Design integration with UVM testbench, QuestaSim tool is used to verify UVM simulation run. The result shown in figure 30 is to prove the previously specified equivalence checking in addition to UVM test-passed simulation of 200030 ns.

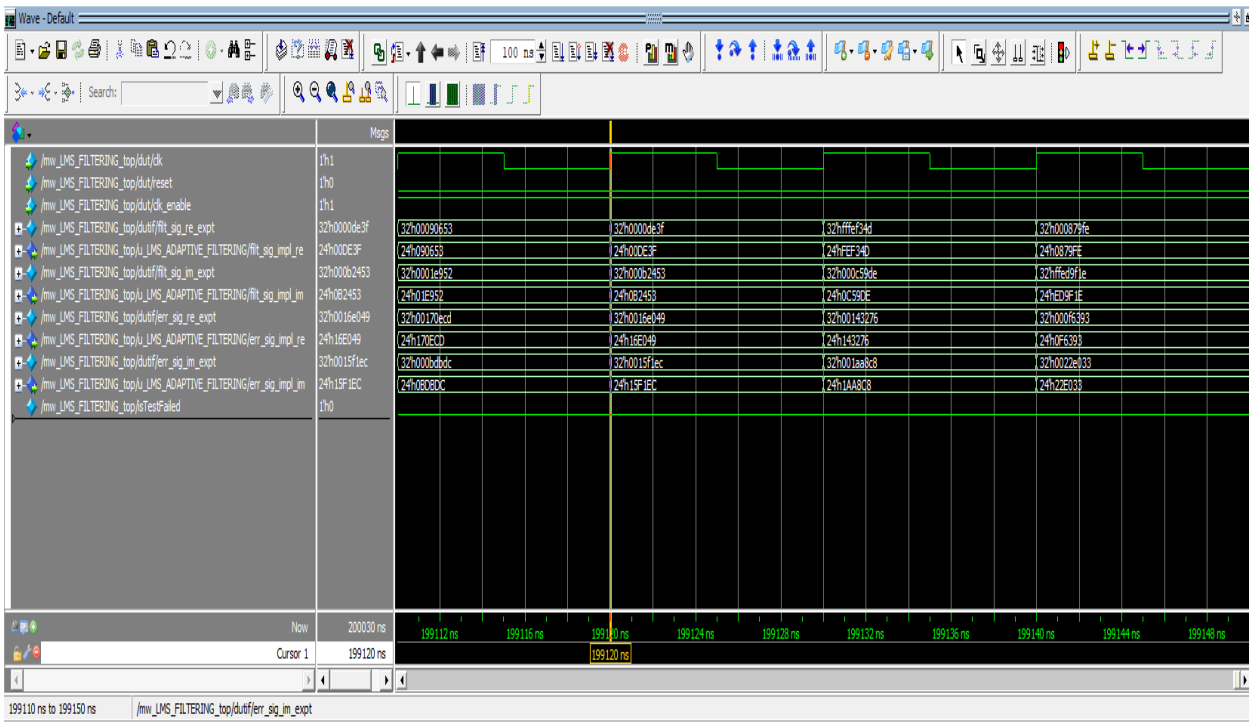

Figure 31. Detailed waveform of inspected signals from UVM Testbench simulation

To analyze UVM test simulation in detail, the signal waveform as seen in figure 31 above is to present all needed output signals over time. From a detailed observation at the yellow cursor, for instance, at 199120 ns, err_sig_re_expt is Simulink DPI-C expected estimation error captured from scoreboard that its value is equal to the value of err_sig_impl_re which is RTL implemented estimation error. Their identical value under hexadecimal format is 16E049, although their data bit width is different. The same results are obtained for filtered output at both data paths.

6.5 System Verilog Assertions

An assertion is a statement about a design's intended behavior which must be verified. It is used to validate the behavior of the system defined as properties which are design behavioral attributes. Its sole purpose is to ensure consistency between the designer's intention, and what is created. [15,3-4.]

For this Thesis project, System Verilog assertions were used to verify the correct functionality of signal logic under valid control. Simple tests written were to check that tap delay processing and weight update logic worked as expected. The register interface writes the configuration and exports the data from the module internal registers to the

top-level configuration and data registers [11,80]. This connectivity from RTL design module to top level registers to be coded as System Verilog assertions binding shown in listing 21 below is to read RTL implemented output data then drive them into assertion properties through the binding interface. Concurrent assertions are used at each rising edge of the valid system clock to evaluate the behavior of RTL signal logic.

```
bind mw_LMS_FILTERING_top.u_LMS_ADAPTIVE_FILTERING.u_LMS_ADAPT_REAL  LMS_ADAPTIVE_FILTERING_SVASRT lms_adapt_sva_inst_re(
 .clk(clk),
 .reset(reset),
 .valid_in(valid_in),
 .Tapped_Delay_reg(Tapped_Delay_reg),
 .noise_signed(noise_re_signed),
 .weights_sig(weights_re),
 .filter_coeff_sigl(filter_coeff_re)
 );

bind mw_LMS_FILTERING_top.u_LMS_ADAPTIVE_FILTERING.u_LMS_ADAPT_IMAG  LMS_ADAPTIVE_FILTERING_SVASRT lms_adapt_sva_inst_im(
 .clk(clk),
 .reset(reset),
 .valid_in(valid_in),
 .Tapped_Delay_reg(Tapped_Delay_reg),
 .noise_signed(noise_im_signed),
 .weights_sig(weights_im),
 .filter_coeff_sigl(filter_coeff_im)
 );
```

Listing 21. HDL entities binding to System Verilog assertion properties

To implement modular System Verilog assertions, the coding style used splits assertion declarations into two parts as can be seen in listing 22. Sequence expressions are to configure the functional tests for equivalence checking under control logic whereas properties are to specify the trigger conditions of the previous configured sequence expressions. To be more specific, from line 21 to line 25, it means when valid_in is activated, weights_sig value assigns to weight_v which will be stored as a temporary variable. At the one next clock cycle, this variable will be compared to fil-ter_coeff_sig_1 which is the weight update logic. The similar configuration applied to tap delay processing as can be seen from line 27 to line 30 that there is a variable assignment from noise_signed to the intermediate variable noise_in_v which will be in comparison with the shifted element of tap delay register after one delayed clock cycle. For property declarations, from line 33 to line 47, it describes that at each rising edge of system clock, when valid_in triggers, previous configured sequence expressions will be verified.

```
2  `include "../DPI_dut/LMS_FILTERING_dpi.sv"
3  `include "../uvm_artifacts/mw_LMS_FILTERING_if.sv"
4  `include "uvm_macros.svh"
5  `define true 1
6  import uvm_pkg::*;
7  import lms_pkg::*;
8
9  module LMS_ADAPTIVE_FILTERING_SVASRT(
10   logic   clk,
11   logic   reset,
12   logic   valid_in,
13   logic signed [23:0] Tapped_Delay_reg [0:43],
14   logic signed [23:0] noise_signed,
15   logic signed [23:0] weights_sig [0:44],
16   logic signed [23:0] filter_coeff_sigl [0:44]
17   );
18
19 //-----sequences---------------//
20
21   sequence weight_assignment;
22     logic signed [23:0] weight_v [0:44];
23
24     (valid_in, weight_v = weights_sig) ##1 (filter_coeff_sigl === weight_v);
25   endsequence
26
27   sequence tapped_delay_check;
28     logic signed [23:0] noise_in_v;
29
30     (valid_in, noise_in_v = noise_signed) ##1 (Tapped_Delay_reg[43] === noise_in_v);
31   endsequence
32 //-----properties---------------//
33   property weight_update_check_p;
34     disable iff (reset)
35     @(posedge clk)
36       valid_in
37       |->
38       weight_assignment;
39   endproperty
40
41   property tapped_delay_check_p;
42     disable iff (reset)
43     @(posedge clk)
44       valid_in
45       |->
46       tapped_delay_check;
47   endproperty
```

Listing 22. Sequence and property declarations for SystemVerilog concurrent assertions

In listing 23, from line 50 to line 62, there are assertions to verify the design properties that if the asserted logic behaviors are not as expected then uvm_error actions will be activated to stop UVM simulation along with the detailed messages shown on HDL simulator display.

```
49  //-----assertions-------------//
50    assert_weight_update: assert property (weight_update_check_p)
51      begin
52        `uvm_info("weight_update_check", "weight update running from high-triggered valid gating", UVM_DEBUG)
53      end else begin
54        `uvm_error("weight_update_check", "weight update process fails ")
55      end
56
57    assert_tapped_delay: assert property (tapped_delay_check_p)
58      begin
59        `uvm_info("tapped_delay_check", "noise input tapped delay running from high-triggered valid gating", UVM_DEBUG)
60      end else begin
61        `uvm_error("tapped_delay_check", "input tapped delay process fails ")
62      end
63
64     //-------------------coverage-----------------------//
65    cover_weight_update_check: cover property (weight_update_check_p);
66    cover_tapped_delay_check: cover property (tapped_delay_check_p);
67
68  endmodule
```

Listing 23. Concurrent assertions and cover properties written by System Verilog

The assertion results shown in figure 32 are all test-passed for each verified procedure under UVM verification.

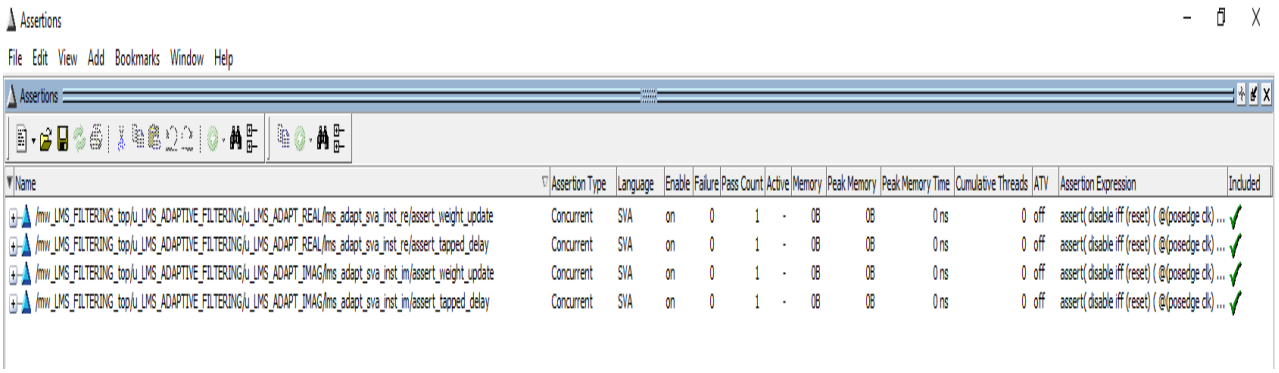

Figure 32. The test-passed results of SV concurrent assertions under UVM verification

In addition, from line 65 to line 66 in listing 23, cover properties are used to guarantee that if the properties do not fail but pass with “cover” then the design does not have a bug associated with that properties and the conditions have been exercised [14,35].

7 Conclusion

The Thesis work presented an adaptive FIR filtering noise cancellation module which smoothly filtered the corrupted noisy signal to recover the clean original sinusoidal signal. The design module implements Least Mean Squared algorithm with filter coefficients update stages to create the compatible weights for each adaptive filtering procedure and eliminate the uncorrelated noise from the corrupted signal. The main design goals for this module were configurability and reusability. These were achieved already in algorithm level with the use of modular coding style to create instantiated functions for partial configurations. The configurability was implemented with the use of Matlab command line interface which provided an interface to the configured parameters. RTL reusability was realized by describing parameters configuration and each used function inside HDL package. The implementation work was done adhering to Mathworks digital design flow.

Verification was likewise done following an ASIC verification flow before physical design period. The verification flow included many modern tools and practices such as the use of a re-usable UVM test bench for module and system level verification, specific use case. The module was verified to produce exactly the same results as the algorithm by comparing the fixed-point data the algorithm generated to the data the RTL model output during functional verification. The data were compared clock accurately bit-to-bit. Additionally, no errors or bugs were found from the design module. the module passed all required functionality tests and was cleared for RTL hand-off. The complexity of the Adaptive FIR Filtering noise cancellation module is quite trivial to dial back and forth as the tapped delay process, smoothing filtering stages are all modular. [11,85.]

The design module serves the purpose of noise cancellation process based on adaptive LMS algorithm with allowed configured difference between the original input and correlated filtered output. The relevant theory related adaptive filter was explored. The theory and mathematical model of the design module were also explained. The design flow followed the familiar top-down model where work is done in subsequent stages. First the algorithms were developed to fixed-point models which physically realizable. Then the RTL micro-architecture of the module was designed on the basis of the algorithms provided. The actual HDL was written following the RTL micro-architecture description. Verification work was started after all functionality was included in the RTL. The verification

environment was an UVM test bench which supported many modern verification techniques including test case establishment, code coverage and System Verilog assertions. [11,87.]

Hardware deployment and implementation and the related challenges, such as IP core generation to be integrated with a register abstraction layer by using AXI protocols, Transaction Level Modeling using SystemC and IP-XACT, could be innovated for future development of this Thesis project, since the design module was already modified to be modular and reusable, so that it has a scalable architecture with generic parameter specifications. This gave some valuable insight because RTL design coding style and configurations originally targeted to an SoC.

The aim of this work was to design, implement and verify a fully adaptive FIR filtering noise cancellation module following Mathworks digital design flow. This objective was successfully reached as the design module met all pre-defined design specifications.

References

1 Simon Haykin. Adaptive Filter Theory. Fifth Edition. Essex, England: Pearson Education; 2014.

2 Michael Parker. Digital Signal Processing 101 Everything You Need to Know to Get Started. Second Edition. Oxford, United Kingdom: Newnes - an imprint of Elsevier; 2017.

3 Samuel D. Stearns, Don R. Hush. Digital Signal Processing with Examples in Matlab. Second Edition. Florida, USA: CRC Press; 2011.

4 Lizhe Tan, Jean Jiang. Digital Signal Processing Fundamentals and Applications. Third Edition. London, United Kingdom: Academic Press – an imprint of Elsevier; 2019.

5 Andreas Antoniou. Digital Signal Processing, Signals, Systems, and Filters. United States of America: The McGraw-Hill Companies; 2006.

6 Uwe Meyer-Baese. Digital Signal Processing with Field Programmable Gate Arrays. Fourth Edition. Florida, USA: Springer; 2014.

7 Vinay K. Ingle, John G. Proakis. Digital Signal Processing Using Matlab. Third Edition. Stamford, USA: Cengage Learning; 2012.

8 Roger Woods, John McAllister, Gaye Lightbody, Ying Yi. FPGA-Based Implementation of Signal Processing Systems. West Sussex, United Kingdom: John Wiley & Sons; 2008.

9 Steven W. Smith. The Scientist and Engineer's Guide to Digital Signal Processing. Second Edition. California, USA: California Technical Publishing; 1999.

10 Richard G. Lyons. Understanding Digital Signal Processing. Third Edition. Boston, Massachusetts, USA: Prentice Hall; 2011.

11 Ville Kivelä. Design, Implementation and Verification of a Digital Predistorter Overdrive Protection Module. Master's Thesis. Oulu, Finland: University of Oulu, Faculty of Information Technology and Electrical Engineering; 2016.

12 Joonas Järviluoma. Rapid Prototyping from Algorithm to FPGA Prototype. Master's Thesis. Oulu, Finland: University of Oulu, Department of Electrical Engineering; 2015

13 Joseph Petrone. Adaptive Filter Architectures for FPGA Implementation. Master's Thesis. Florida, USA: The Florida State University College of Engineering, Department of Electrical and Computer Engineering; 2004.

14 Ashok B.Mehta. SystemVerilog Assertions and Functional Coverage, Guide to Language, Methodology and Applications. Third Edition. Cham, Zug, Switzerland: Springer; 2020.

15 Harry D. Foster, Adam C. Krolnik, David J. Lacey. Assertion-based Design. Second Edition. Boston, Massachusetts, USA: Kluwer Academic Publishers; 2004.

16 Janick Bergeron. Writing Testbenches Using System Verilog. New York, USA: Springer; 2006.

17 Mentor's Verification Academy. Universal Verification Methodology UVM Cookbook. Mentor Graphics Corporation; 2018.

18 Mentor's Verification Academy. Coverage Cookbook. Mentor Graphics Corporation; 2019.

19 ANSI/IEEE Std 1076-1993. IEEE Standard VHDL Language Reference Manual. New York, USA: IEEE; 1994.

20 IEEE Std 1800-2005. IEEE Standard for System Verilog - Unified Hardware Design, Specification, and Verification Language. New York, USA: IEEE; 2005.

21 Accellera Systems Initiative. Universal Verification Methodology (UVM) 1.2 Class Reference. California, USA: Accellera; 2014.

22 Intel Corporation. Intel Quartus Prime Standard Edition User Guide: Design Recommendations UG-20175. [Online]. USA: Intel; 24 September 2018. URL:https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-qps-design-recommendations.pdf [Accessed on November 9th 2020]

23 MathWorks. Verification with Cosimulation. [Online]. URL: https://www.mathworks.com/help/hdlverifier/hdl-verification-with-cosimulation.html [Accessed on November 9th 2020]

24 MathWorks. LMS Filter. [Online]. URL: https://www.mathworks.com/help/dsp/ref/lmsfilter.html [Accessed on November 9th 2020]

25 Henry Chan. UVM Spells Relief - Create Robust Test Environments with Ease. [Online]. Aldec Blog. URL: https://www.aldec.com/en/company/blog/110--uvm-spells-relief [Accessed on November 9th 2020]

26 MathWorks. HDL Verifier. Test and Verify Verilog and VHDL Using HDL Simulators and FPGA boards. [Online]. URL: https://www.mathworks.com/products/hdl-verifier.html#resources [Accessed on November 9th 2020]

27 Roshny Jose George. Design of An Adaptive Filtering Algorithm for Noise Cancellation. Volume 2 Issue 4. [Online]. IRJET; July 2015. URL: https://www.irjet.net/archives/V2/i4/Irjet-v2i4105.pdf [Accessed on November 9th 2020]