A SIMULATION BASED METHODOLOGY FOR THE DEVELOPMENT OF EMBEDDED-ANALOGUE-MIXED-SIGNAL SYSTEMS USING SYSTEMC-AMS

Benjamin Mulwa Kathale
I56/72438/2008

A thesis submitted in partial fulfillment for the degree in Master of Science in Physics of University of Nairobi

August 2016

DEDICATION

To my treasured family, wife – Josephine, son – Samuel and daughter - Lydia

DECLARATION

This thesis is my original work and has not been presented for award of any degree in any other university.

Signature _____ Date_____

Benjamin Mulwa Kathale

University of Nairobi

This thesis has been submitted for examination with my approval as university supervisor.

Signature _____ Date_____

Mr. A.C.K. Mjomba

Department of Physics

University of Nairobi

Signature _____ Date_____

Dr. K.A. Kaduki

Department of Physics

University of Nairobi

ABSTRACT

The development of embedded analog mixed signal systems has been a challenge especially in understanding the specifications of the required components and functionality of the system under development during the development process. In addition, to understand the interaction of components operating with mixed signals has been a problem since there has been no way in which this can be handled effectively.

Since the development of electronic systems start from the functional down to the implementation, a methodology that can be used in all the development levels is required. In this work, a methodology that can be used to model and simulate embedded analogue mixed signal systems has been developed. This methodology, referred to as Model-Simulate-Refine-Synthesis (MSRS), has three stages – functional, non-functional and implementation. The three stages help the system developer to model and simulate in the three levels of system development.

The MSRS methodology has been tested with two cases that are modeled and simulated using SystemC-AMS. The first case is the modeling and simulation of an oscilloscope. This case demonstrates the application of the methodology where analogue signals of different frequencies are sampled and displayed as digital waveforms. Among other things, architectural exploration seamlessly flows into implementation. In the second case, the same methodology has been used to model and simulate a signal generator. The simulation results are compared with a similar signal generator implemented in a Fusion FPGA. The waveforms produced by the Fusion FPGA are replicated by the simulated signal generator.

TABLE OF CONTENTS **PAGE**

APPENDIX B: SIGNAL GENERATOR CODES

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF ACRONYMS

ADC	-	Analog-to-Digital-Converter
AMS	-	Analog-Mixed-Signal
CA	-	Cycle-Accurate
CAD	-	Computer Aided Development
CS	-	Capture-and-Simulate
DAC	-	Digital-to-Analog-Converter
DE	-	Discrete Event
DS	-	Describe and Synthesis
DT	-	Discrete Time
DUT	-	Device Under Test
E-AMS	-	Embedded Analog Mixed Signal
ECAP	-	Electronic Circuit Analysis Program
ELN	-	Electrical Linear Network
FPGA	-	Field Programmable Gate Array
FSM	-	Finite State Machines
HDL	-	Hardware Description Language
HDLs	-	Hardware Description Languages
HW	-	Hardware
IEEE	-	Institute of Electrical and Electronics Engineering
IP	-	Intellectual Property
LSF	-	Linear Signal Flow
LUT	-	Look Up Table
MoC	-	Models of Computation
MSB	-	Most Significant Bit
MSRS	-	Model-Simulate-Refine-Synthesis
PCAM	-	Pin Cycle-Accurate-Model
PSM	-	Process State Machines
RC	-	Resistor-Capacitor
RTL	-	Register Transfer Level

SER	-	Specify-Explore-Refine
SL	-	System Level
SPICE	-	Simulation Program with Integrated Circuit Emphasis
SW	-	Software
TDF	-	Timed Data Flow
TLM	-	Transaction Level Modeling
VCD	-	Value Change Dump
VCO	-	Voltage Controlled Oscillators
VHDL	-	Very High Speed Integrated Circuit Hardware Description Language

ACKNOWLEDGEMENT

I am deeply indebted to a number of people who in one way or another helped during the period of this work:

First, I wish to thank my supervisors, Mr. A. C. K. Mjomba and Dr. K. A. Kaduki for their continued guidance and direction during the entire period of the research. The time they spent guiding me all through and carefully reading and making suggestions on the manuscript at its various stages deserves special mention.

Many thanks to my colleagues Mr. Walter Maina and Mr. Elias Gitau who gave useful suggestions on the installation of SystemC, SystemC-AMS and other related programs is highly appreciated. The positive contribution and discussions with Mr. Elias Gitau on the initial stages of the working on SystemC-AMS deserves appreciation.

The support accorded by the technical staff of the Department of Physics and the entire department is highly valued.

I owe much to my treasured family especially my dear wife, Josephine, and friends who personally encouraged and prayed for me especially when things did not seem to go well. To my dear pastor, Jonathan Nzola, his wife, Jenifer Ivuso, all the church members from JCC-Tassia, Pst. Samuel Mutuku and Pst. Daniel Nzyuko who stood with me in prayer, I am grateful.

CHAPTER ONE

INTRODUCTION

1.1 Background of the study

An Embedded Analog Mixed Signal (E-AMS) system is an electronic system integrated into a device or an appliance, aiming at making the behavior of the device more intelligent. It makes the device or appliance in question easier to operate or use, more energy efficient, safer, friendlier for the environment, and or perform better, (Broedes, 2010). To enlarge the flexibility and the maintainability, most E-AMS not only contain hardware but also contain software components (Mahne, 2011). The hardware components are either analog, digital or even both, depending on the requirements of the system. In most cases, the components are implemented as multi-processor systems on a single chip while others are a collection of discrete chips interconnected to form the embedded system.

Developing E-AMS is generally complex. Gajski and Kuhn offer a way to deal with this complexity. In their approach, they propose four levels of domain abstractions and three levels of system development abstractions as shown in Figure 1.1 (Gajski & Kuhn, 1983). The Y-Chart explains the differences between different development tools and different development methodologies in which these tools are used.

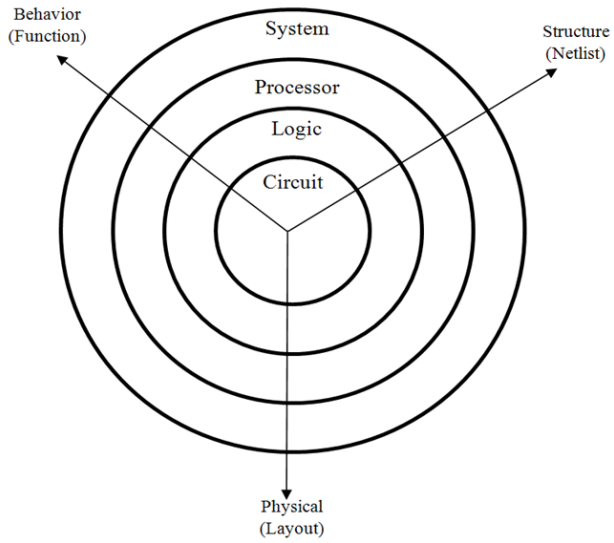


Figure 1.1: The Y-Chart (Gajski & Kuhn, 1983)

Behaviour is a representation of the system as a black box. This gives the functionality of the system where its outputs are described in terms of its inputs over a given time. This representation does not indicate in any way how the black box is build box or its structure. The stucture development presents the black box as a block diagram. The block diagram can be decomposed into a set of components and connections. The physical development brings in the element of dimensions to the structure. It specifies the physical size and the relative position of each component as well as the port and connection on the chip, printed circuit board or any other container.

For the purpose of domain abstractions, the Y-Chart also empasizes circuit, logic, processor and system levels of abstraction shown by the concentric circles in Figure 1.1. The names of the abstraction levels are related to the kind of components generated at each particular level. This means that at circuit level, discrete components such as transistors, capacitors e.t.c. are generated. Logic gates and flip-flops are generated at logic level, special hardware components such as memory controllers at processor level and embedded systems consisting of processors, memories and other components generated at the system level. This framework provides a good starting point for system development. On top of this framework, methodologies have been developed to support modeling, automation and simulation (Gajski & Kuhn, 1983)

Automation is most effective on standard E-AMS development. However automation relies on preconceived models and model transformations thus limiting innovation. On the other hand, modeling and simulation does not suffer those limitations. It is in this respect that a methodology based on modeling and simulation is chosen to be developed. Simulation of E-AMS requires the support of Analogue Mixed Signal (AMS) behavior at each level of abstraction. Due to this reason, the modeling formalisms should be based on models of computation that support the AMS behavior.

In addition to modeling, architectural exploration is usually carried out in a simulation context. It involves the following steps: evaluation of the simulation results against specifications and adjustment of specifications. This may in turn lead to modification of architectural model. This is carried out for both functional and non-functional specifications.

System simulation requires developing and executing all the components of a system using a computer program before implementation of the system. Before the introduction of system simulation, developers used to development and fabricate the system before testing its functionality. This would lead to abandoning the system and developing a new system again in cases where the system fails to function as anticipated. This leads to wastage of resources and prolonged time to market (Gajski *et al.*, 2009) (Barnasconi, *et al*., 2010). System simulation and analysis helps the developer to ascertain the functionality of the system and the specifications of the components before implementation. For innovative systems, the behavior of the system may not be well understood. Simulation provides a preview of at least some of the behavior of the system.

At the heart of every simulation is a simulation language. Some languages such as Electronic Circuit Analysis Program (ECAP) could simulate only hardware analog components (Jensen, 1966), (Roberts & Harbourt, 1967). Simulation Program with Integrated Circuit Emphasis (SPICE) was developed and was capable of simulating and analyzing discrete analog components at circuit level of abstraction (Saxena, *et al.*, 2012), (Dowell, 2011). Hardware Description Languages (HDLs) were developed to simulate discrete signals at Register Transfer Level (RTL) of abstraction (Gajski, *et al.*, 2009), (Pedroni, 2004). These HDLs included Very High Speed Integrated Circuit Hardware Description Language (VHDL) (Ashenden, 2010) and Verilog. Later analog extensions were incoperated in VHDL (VHDL-AMS) and Verilog (Verilog-A) to handle analog hardware components simulation (Vachoux, 1998), (Zorzi, Franzk, & Speciale, 2003), (Thomas & Moorby, 2002), (Szermer, Daniel, & Napieralski, 2003). Although VHDL, Verilog, VHDL-AMS and Verilog-A could simulate both analog and digital hardware components (Pecheux, *et al.*, 2005), the software components required in embedded systems were simulated using C and C++ languages (Black & Donovan, 2004). This led to failure of a common test bench which could be used in the simulation of both hardware and software components since the two categories of the simulation languages are different. Without a common test bench, the functionality of the hardware and software components of an embedded system could not be determined precisely.

A class library within C++ called SystemC was developed to be used in simulation of digital hardware components and software components of an embedded system (Black & Donovan, 2004). SystemC simulation kernel was not developed to handle modeling and simulation of analog and continuous-time systems but supports Transaction Level Modeling (TLM) (Cai & Gajski, 2003), (Ghenassia, 2005). TLM allows the developer to perform abstract modeling, simulation and development of discrete-event Hardware/Software (HW/SW) system architectures (Barnasconi *et al.*, 2010), (Donlin, 2004). Although SystemC provides support for digital hardware and software integration simulation tests, it does not support analogue simulation tests. Therefore, SystemC lacks the support of describing analog behavior of embedded systems. Due to this challenge, Analog-Mixed-Signal (AMS) extensions has been introduced in SystemC (Barnasconi, *et al.*, 2010) to generate SystemC class library called SystemC-AMS.

SystemC-AMS extensions define language constructs with execution semantics for mixed-signal systems. The class definitions provided by the AMS language standard form the basis for the creation of a C++ class library implementation, used in combination with an Institute of Electrical and Electronics Engineering (IEEE) 1666-2005 compatible SystemC implementation. Such an implementation is used to create AMS system-level models to build an executable specification, to validate and optimize the E-AMS system architecture, to explore various algorithms, and to provide the development team with an operational virtual prototype of the entire E-AMS system. To support these use cases, the SystemC-AMS extensions define modeling formalisms to model E-AMS system-level behavior at discrete-time and continuous-time levels of abstraction (Barnasconi, *et al.*, 2010). The modeling formalisms include; Timed Data Flow (TDF), Linear Signal Flow (LSF), Electrical Linear Networks (ELN), Discrete Event (DE) and Discrete Time (DT). These modeling formalisms have defined execution semantics and therefore serve as models of computations (MoC). They guide development of executable E-AMS models.

1.2 Problem Statement

There is a growing trend for tight interaction between embedded HW/SW systems and their analog physical environment. Consequently, systems in which digital HW/SW components are

interwoven with analog and mixed-signal components are realized. Such systems are called embedded systems or embedded analog mixed-signal systems (E-AMS). Developing the E-AMS systems becomes a challenge while trying to understand the interaction between the HW/SW and the analog and mixed-signal subsystems at the architectural level. Methodologies have been developed which are used to model and simulate these E-AMS system but they have some limitations. The limitations include lack of a common design tools to model both hardware and software components. They also lack the capability of modeling and simulation of E-AMS from the functional down to the implementation level. Due to these, new ways to develop the interacting HW/SW subsystems and the mixed-signal subsystems are required. SystemC-AMS provides an appropriate tool to support the modeling and simulation of interactions spanning functional level down to the implementation level and inclusive of software and both analogue and digital hardware (Barnasconi, *et al*., 2010), (Mähne, 2011). Therefore, a methodology that builds on these capabilities of SystemC-AMS is developed.

1.3 Objectives

1.3.1 General Objective

The general objective of this work was to generate a methodology for modeling and simulation of Embedded Analog Mixed Signal (E-AMS) systems.

1.3.2 Specific Objectives

1. To generate functional, non-functional and system implementation models, each for its corresponding level in the design process
2. To verify the functioning of each model through simulation process
3. To validate the models by comparing its output with the stakeholders expectations
4. To test the methodology through modeling and simulating an oscilloscope and a signal generator as case studies

1.4 Justification and Significance

System development process gives better results if a methodology that helps the developers to model and simulate the system under development at all levels (functional level, architectural level and implementation level) and including both hardware (analog and digital) and software

components is used (Barnasconi, *et al*., 2010). Gajski *et al*, (2009) presents some methodologies which were developed to model and simulate electronic systems. These methodologies could make the system developers model and simulate the HW and SW components of the systems separately due to lack of a common design tools. This could lead to the repeat of the design process if the HW and SW components failed to match and give the expected results.

Also, the methodologies could help the system developer in modeling and simulation of the systems from the functional level down to the TLM level but not up to the implementation level. This would make the system developers repeat the design process if errors would occur at the implementation level where modeling and simulation was not being done.

Therefore, in response to these limitations, this work provides system developers with a methodology that they can use in their development work to model and simulate analog, digital and software components concurrently. It also supports modeling and simulation from the system functional level down to the system implementation level.

Further, the methodology readily supports architectural exploration, modeling and simulation seamlessly. This has been made possible by the use of SystemC-AMS as a support tool for this methodology.

CHAPTER TWO

LITERATURE REVIEW

2.1 Introduction

Modeling and simulation is one of the fundamental processes required in development of E-AMS systems. This chapter presents prior work done by different researchers on modeling and simulation methodologies used while developing E-AMS systems.

2.2 Related research work

In the development of E-AMS systems, modeling and simulation play a very important role in the development process. They are used to verify the functionality and properties of the whole system (Ptolemaeus, 2014), (Topper & Horner, 2013), (Karnane, Curtis & Goering, 2009) and further to that is part of the development process (Jeruchim, Balaban, & Shanmugan, 2006). Depending on the modeling and simulation tool, the simulation code can be used to verify the system and be downloaded into a Field Programmable Gate Array (FPGA) board for the implementation (Gajski *et al*., 2009). There have been different development methodologies since 1960s that explain the development process where simulation is involved (Sinha, *et al.*, 2001). Gajski *et al.* (2009) explains the evolution of the development methodologies since early 1960s. In their discussion, three methodologies are presented which include Capture-and-Simulate (CS), Describe-and-Synthesis (DS) and Specify-Explore-Refine (SER) methodology.

In CS methodology, the software (SW) and hardware (HW) development was separated by a system gap. The system gap was a result of the SW developers and the HW developers using different tools. This gap was made worse by the fact that the developers could not work concurrently. The SW developers used to test algorithms and write the requirements document and the initial specification of the system under development. The specification used to be given to the HW developers to begin the hardware development. The HW developers used to start working with a block diagram based on the specification. In this case, the HW developers could not know if their development would meet the requirements on the specification until the gate-level development was produced and simulated. Once the gate net-list was captured and

simulated, developers could determine whether the system worked as specified or not. Sometimes, the system did not work and the specification was changed to accommodate the desired behavior. Figure 2.1 shows a schematic diagram for the CS methodology.

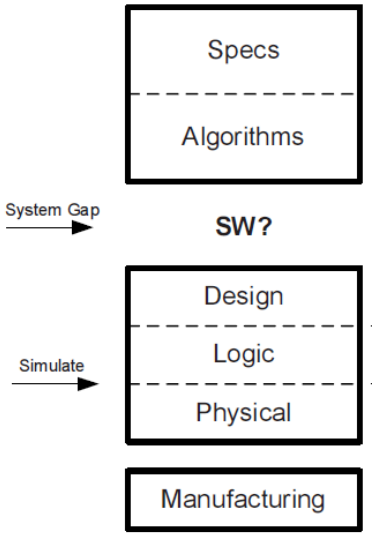


Figure 2.1: Capture-and-Simulate Methodology (Gajski *et al.*,2009)

In 1980s, tools for logic synthesis were brought and significantly altered development flow by capturing both behavior and structure of the development at the logic level. The system developers specified first what they wanted in Boolean equations or Finite State Machines (FSM) descriptions and then the synthesis tools generated the implementation in terms of logic-level net-lists. This formed the DS methodology presented in Figure 2.2. In this methodology the logic level net-list were automatically generated. Now, behavior could be simulated at function/specifications model and then after synthesis at the logic level net-list. This made it possible to verify by simulation the descriptions equivalence between function level and the gate level net-list. The FSM level of abstraction is however too low. It results in huge number of states for even moderately sized systems. In the late 1990s, the logic level had been abstracted to the RTL with the introduction of Cycle-Accurate (CA) modeling and synthesis. Even with the introduction of the two levels of abstraction (RTL and Logic) and two modeling abstractions on each level (behavioral and structural), the system gap found in the CS methodology was still existing since there was no relation between RTL and higher system level.

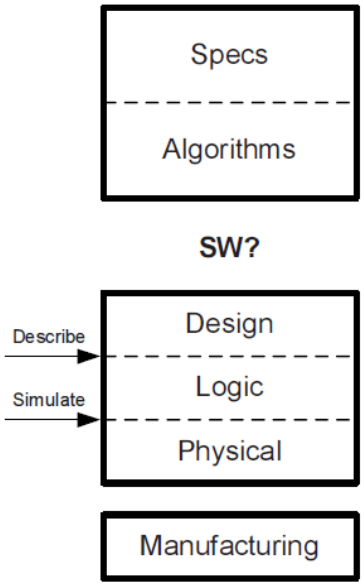


Figure 2.2: Describe-and-Synthesis Methodology (Gajski *et al.*,2009)

Specify-Explore-Refine (SER) methodology presented in Figure 2.3 was introduced in early 2000s and is still in use today. This methodology was aimed at closing the system gap existing in the CS and DS methodologies. To close the gap, a level of abstraction higher than both HW and SW level called the system level was devised and a methodology introduced that incorporates both SW and HW. Behavioral models are first devised at the System Level (SL). They are then synthesized to structural models and then simulated.

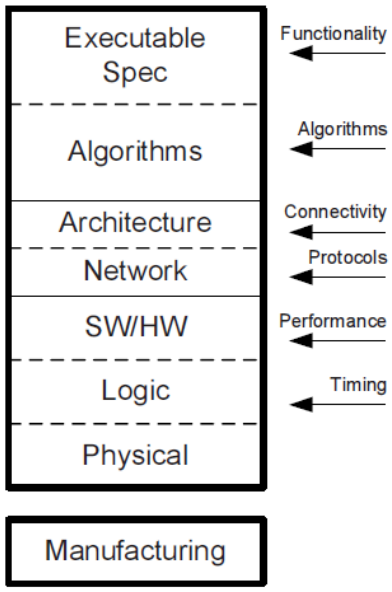


Figure 2.3: Specify-Explore-Refine Methodology (Gajski *et al.*,2009)

The structural model is synthesized further to Cycle-Accurate (CA) model. The CA model represents the whole system under development and can be downloaded into an FPGA board or a microcontroller board using standard Computer Aided Development (CAD) tools provided by the board suppliers (Calva, *et al*., 2012). This leads to generation of system prototype. If all the synthesis and refinement tasks are automated, the prototype can be generated in a few weeks (Gajski, *et al.*, 2009). This helps the system developer to save on time and effort (Najy, 2013), (Kodi, 2008). In addition, since the simulation helps the developer to ascertain the functionality of the system and the properties of any component to be used before implementation, then the system developer would not waste on resources by implementing a system with errors, which has to be abandoned.

Besides the introduction of SER methodology and the CA model used in the implementation of the system prototype, simulation as part of the development process has other benefits as presented by different researchers. In a research done by Kelemenova *et al*. (2013), simulation plays a major role in performing development tradeoffs of the behavioral models of the mixed signal components of the system under development. This may be hard or even impossible to do on paper-based development reviews. Further, they emphasized on the point that the models used in the simulation, can be reused and elaborated to build and test more detailed developments. In their discussion, it comes out clear that the models used in the simulation can become the development artifacts from which hardware developers automatically generate Hardware Description Language (HDL) code. This is also presented by Mischkalla *et al.* (2010). Kelemenova *et al.* (2013) explain that, the developers learn more about system dynamics through simulation than from real systems because it provides details on some properties such as force, current, torque (just to mention but a few).

Mathematical modeling and simulation tools provide an efficient approach for predicting operational behavior, correcting development errors, eliminating prototyping steps and reducing system components through component tradeoffs (Montealegre *et al.*, 2013). This is essential to study the impact of cost and development modifications of E-AMS systems. Wilson and Mantooth (2013) explain that simulation of E-AMS helps the developer in analyzing many scenarios of electronic systems. It is also used in optimization of statistical context to evaluate

many variations and, further to that, helps in troubleshooting of the system as well as helping the developer to look inside the components within a development to enhance the developer's understanding of the behavior of the system.

To synopsize, it has been pointed out that the introduction of SER methodology was to close the system gap by rising of levels of abstraction to system level. Though the system gap problem was solved after introducing the SER methodology, it was only for the modeling and simulation of digital systems (Gajski, *et al.*, 2009). E-AMS systems however embrace both analogue and digital subsystems. It is in this respect that the methodology is developed to provide a seamless modeling, architectural exploration and simulation support for E-AMS development.

CHAPTER THREE

METHODOLOGY

3.1. Introduction

The development of any E-AMS system requires modeling, architectural exploration and simulation. This chapter presents a modeling, architectural exploration and simulation methodology that forms the basis of this work. A simple methodology that is similar to the one developed is presented in section 3.2. The purpose of section 3.2 is to provide a background helpful to understanding the Model-Simulate-Refine-Synthesis (MSRS) methodology that follows in section 3.3. Finally, in section 3.4, a modeling and simulation language called SystemC-AMS is presented. The modeling and simulation capabilities of this language are essential to the MSRS methodology.

3.2. System Level Development Synthesis Methodology

Gajski *et al* (2009) presents a system level development methodology that describes systems from behavioral level down to the architectural level. The methodology begins by describing the behavior of the system, which is presented as some model of computation such as a set of sequential and parallel processes communicating through message passing channels. The model executes on a platform defined after estimating some characteristics of the application. This executable code is also called application code. After defining the platform, it is partitioned and each partition assigned to a processor or Intellectual Property (IP) in the platform. The Model of Computation is mapped on the platform and a model that can be simulated is then generated. The model is simulated to verify that the application executes on the platform and satisfies the requirements of the system. This model is called the Transaction Level Model (TLM).

After simulation, the results of the simulation are evaluated from which development optimization is carried out on the platform and/or the application code. It is also possible to change the mapping of the application to the platform. After satisfying the application code, platform and the mapping, each component is synthesized to generate an implementation level model. This model is called Pin Cycle-Accurate-Model (PCAM). The PCAM contains binaries

for downloading to processors and RTL descriptions for the HW parts in the platform. The PCAM is downloadable to standard FPGA boards for system prototyping whose results can be used for final optimization of the whole development. The methodology is presented in Figure 3.1.

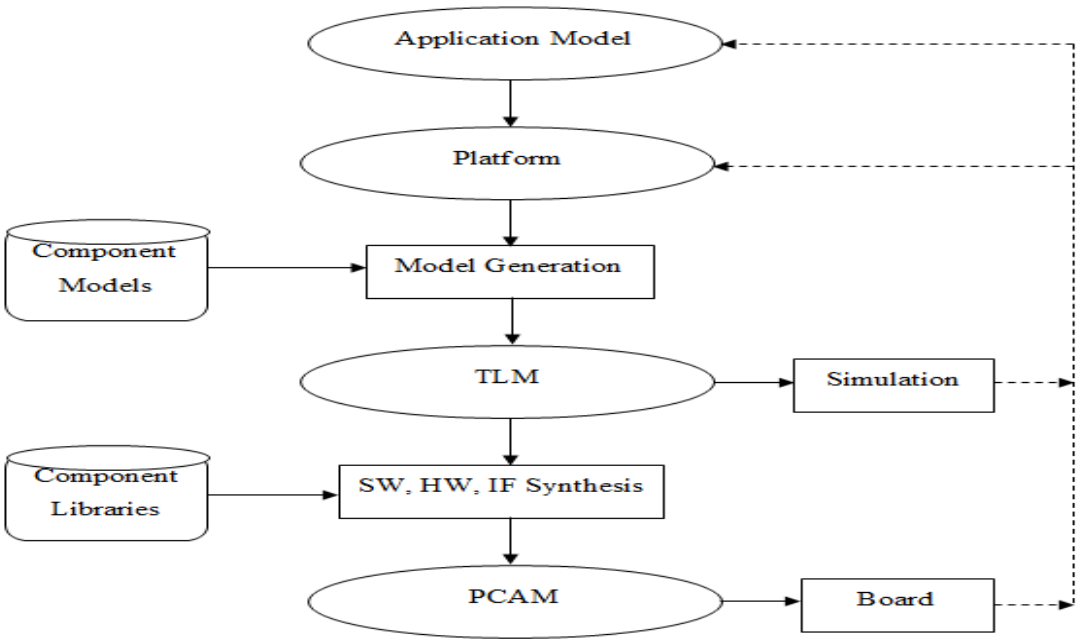


Figure 3.1: System Level Development Synthesis Methodology (Gajski, *et al.*, 2009)

In the methodology presented in Figure 3.1, simulation is done only when TLM is generated. This approach may lead to development time wastage. If the development components are not meeting all the requirements, then the whole process has to be started again from the application model code, mapping and generation of the TLM. In addition, no simulation has been suggested for the PCAM model. This means that the errors in the model are detected during testing. While not all errors are detected by simulation, it is usually more economical to catch as many errors as possible at the earliest opportunity.

Due to these limitations of the system level development methodology, Model-Simulate-Refine-Synthesis (MSRS) methodology is developed. In this methodology, simulation and refining are done at every stage before moving to the next stage. The advantage of this methodology is that

the developer moves to the next stage only when sure of the current stage. This leads to saving development time. The developed methodology is presented in section 3.3.

3.3 Model-Simulate-Refine-Synthesis Methodology

The Model-Simulate-Refine-Synthesis (MSRS) is a methodology for developing E-AMS where modeling, simulation and architectural exploration are carried out. The methodology generates three main models, which include Functional model, Non-Functional model and Implementation model. The general view of the methodology is presented in Figure 3.2 and its details discussed in sections 3.3.1, 3.3.2 and 3.3.3.

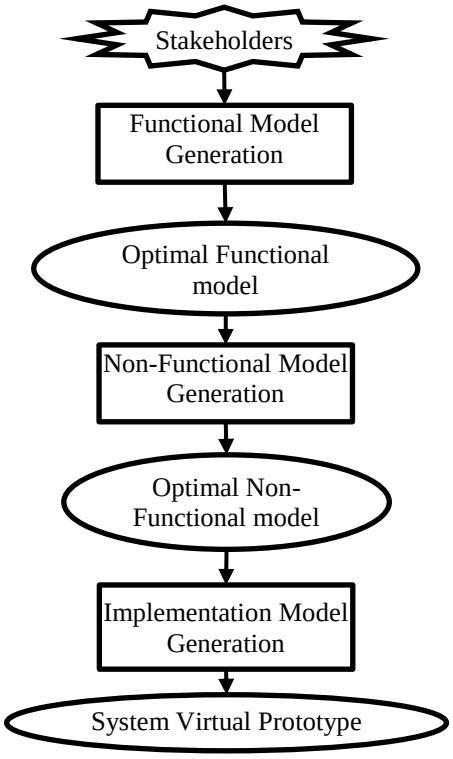


Figure 3.2: Model-Simulate-Refine-Synthesis Methodology

3.3.1 Functional Model Generation

The functional model is in the top-most level of the system development. In the functional level, stakeholders provide their expectations on the output of the system, i.e., the functional attributes of the system under development. Since the methodology follows an innovative approach, it is

not required of the developer and the stakeholders to specify accurately the system functional requirements since they may change during the development process. The components needed for the development of the system are not of much interest – what matters most is the functionality of the system. Consequently, the components of the system are presented as black boxes whose input output behaviors are specified.

The black boxes are modeled as functions when outputs are independent of previous inputs (i.e stateless machine) and state machines when outputs depend on previous states. The black boxes are then combined to form the application model of the system (Barnasconi, *et al*., 2010). The application model is simulated and the results compared with what is expected of the system under development. If the simulation results are not matching with the expected results, then refinement is done. During refinement, if major changes related with the system functionality are required, then the stakeholders are consulted to give their views and contributions. On the other hand, if no major changes are required, then, the functional model is refined further by refining the database input to the model generation, to give refined model.

Once the simulation of the application model yields satisfying results on the functionality of the system under development, an optimal functional model is generated. A block diagram for generating the optimal functional model is presented in Figure 3.3.

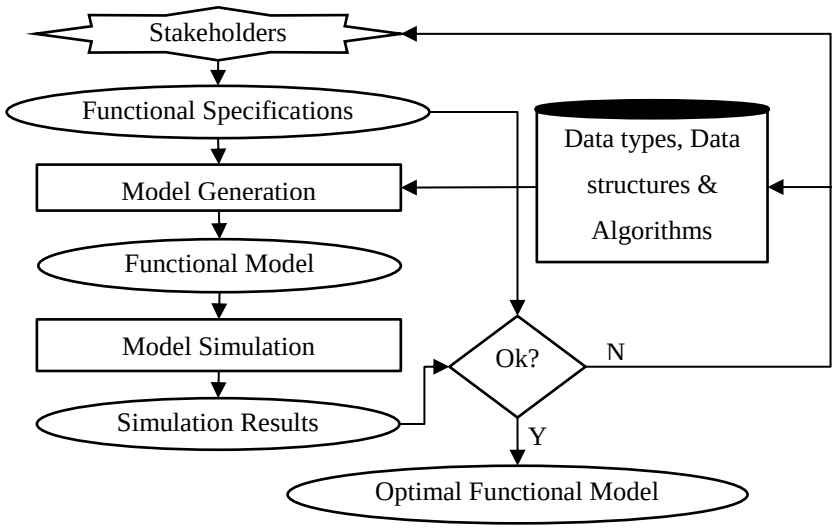


Figure 3.3: Optimal Functional Model Generation

3.3.2 Non-Functional Model Generation

The non-functional model is generated by synthesizing the functional model through combining the non-functional attributes from the stakeholders and knowledge related to architectural styles, patterns, components, platforms and tools. The non-functional model is simulated and refined to the expectations of the stakeholders and an optimal non-functional model is generated. The block diagram of generating the optimal non-functional model is presented in Figure 3.4.

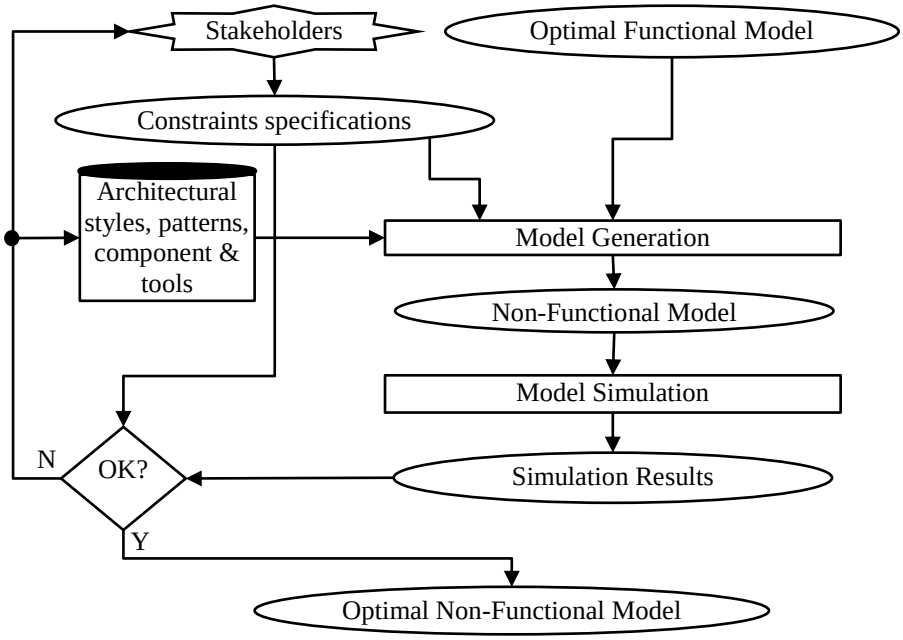


Figure 3.4: Optimal Non-Functional Model Generation

3.3.3 Implementation Model Generation

The optimal non-functional model is combined with the technologies from the HW and SW component technologies and tools to synthesize the system implementation model. In the synthesis, components and subsystems are integrated under a technology specific style or pattern. Before the integration, the interfaces of all components and subsystems must be modeled accurately and the interfaces and data types used in the models should match the actual functionality expected. The abstract communication used within the application models are replaced with concrete signals such as electrical voltages and currents or digital signals. Generic

electrical nodes, pins and buses can be used for analog and digital circuits respectively. These can be made implementation specific if the need arises.

The implementation components are varied. For example some are analogue while others are digital. Component behaviors are also varied. Some may be discrete time or event driven, while others may be continuous time. Thus, different MoCs such as Process State Machines (PSM), Linear Signal Flow (LSF), Timed Data Flow (TDF), Electrical Linear Network (ELN) and Discrete Event (DE) are required for modeling of component behavior. Once the components are modeled, they are integrated to form the system implementation model. The system implementation model is simulated and refined until it gives desirable results. At this point, the system developer obtains refined system implementation model, which can also be referred to as the system virtual prototype. The block diagram of generating the implementation model is presented in Figure 3.5.

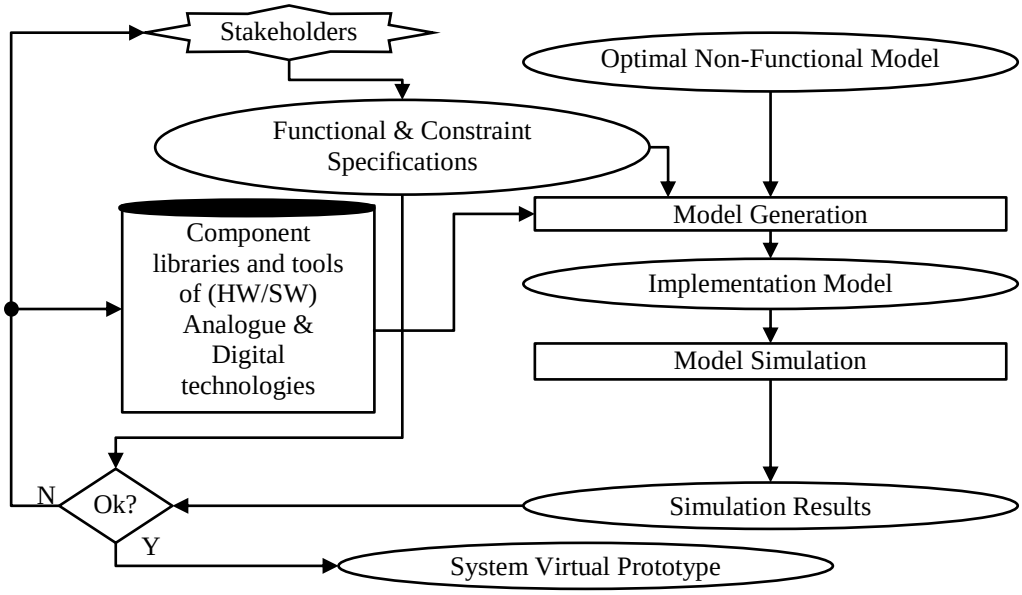


Figure 3.5: System Virtual Prototype Model Generation

The modeling, architectural exploration and simulation of E-AMS systems following the Model-Simulate-Refine-Synthesis methodology presented is supported by SystemC-AMS discussed in section 3.4.

3.4. SystemC-AMS

SystemC-AMS is an extension of SystemC, which can be used in modeling, and simulation of electronic systems up to the register level. The systems that can be modeled and simulated up to register level are only digital systems. Since E-AMS have both analogue and digital components, SystemC cannot be used to simulate the analogue components. Due to this limitation, SystemC-AMS, which has a capability to model and simulate E-AMS systems, was developed. SystemC-AMS has some strengths that makes it a better tool to be used with the developed MSRS methodology.

3.4.1 The strengths of SystemC-AMS

SystemC-AMS can work with five primitive MoCs, which help it to model and simulate any E-AMS component. The MoCs are Linear Signal Flow (LSF), Timed Data Flow (TDF), Electrical Linear Network (ELN), Discrete Time (DT) and Discrete Event (DE) (Barnasconi, *et al*., 2011), (Vachoux, *et al.*, 2004).

The LSF MoC allows modeling of AMS behavior which is defined as relations between variables of a set of linear algebraic equations and is mostly used at the functional level since the system components are presented as functions. On the other hand, TDF MoC considers data as signals sampled in time. The signals are tagged at discrete points in time, which carry discrete or continuous values like amplitudes. This MoC provides behavior descriptions between analogue and digital descriptions and is used at the architectural level where component architectures are modeled. Similarly, the ELN MoC introduces the use of electrical primitives and their interconnections to model the system electrical components. The ELN is mostly used at the implementation level (Barnasconi, *et al*., 2010), (Vasilevski, *et al*., 2007). DE and DT MoCs, which are purely found in SystemC, are developed to model and simulate digital components (Black & Donovan, 2004).

It is important to remember that SystemC models and simulates digital components and SystemC-AMS extensions model and simulate both analogue and digital components. Due to this fact, SystemC-AMS provides seamless modeling and simulation of the analog and digital components found in E-AMS systems.

SystemC-AMS has converter classes that enable systems with mixed signals to be modeled and simulated. The converter classes are used to transit from one MoC to another hence helping them to communicate. The converter classes are summarized in Tables 3.1, 3.2, 3.3 and 3.4.

Table 3.1: DE to TDF, LSF and ELN MoCs Converter Classes

Signal converter Class	Interconnected modules	
	Signal in	**Signal out**
sca_tdf::sca_de::sca_in	DE signal	TDF signal
sca_lsf::sca_de::sca_source	DE signal	LSF signal
sca_eln::sca_de::sca_vsource	DE signal	ELN voltage signal
sca_eln::sca_de::sca_isource	DE signal	ELN current signal

Table 3.2: TDF to DE, LSF and ELN MoCs Converter Classes

Signal converter Class	Interconnected Modules	
	Signal In	**Signal Out**
sca_tdf::sca_de::sca_out	TDF signal	DE signal
sca_lsf::sca_tdf::sca_source	TDF signal	LSF signal
sca_eln::sca_tdf::sca_vsource	TDF signal	ELN voltage signal
sca_eln::sca_tdf::sca_isource	TDF signal	ELN current signal

Table 3.3: ELN to TDF and DE MoCs Converter Classes

Signal converter Class	Interconnected Modules	
	Signal In	**Signal Out**
sca_eln::sca_tdf::sca_vsink	ELN voltage signal	TDF signal
sca_eln::sca_tdf::sca_isink	ELN current signal	TDF signal
sca_eln::sca_de::sca_vsink	ELN voltage signal	DE signal
sca_eln::sca_de::sca_isink	ELN current signal	DE signal

Table 3.4: LSF to DE and TDF MoCs Converter Classes

Signal converter Class	Interconnected Modules	
	Signal In	**Signal Out**
sca_lsf::sca_de::sca_sink	LSF signal	DE signal
sca_lsf::sca_tdf::sca_sink	LSF signal	TDF signal

It should be worth of note that there are no direct converter classes from ELN MoC to LSF MoC and vice versa. This does not mean that there is no communication between the two MoCs since communication can be achieved through the TDF MoC. Therefore, to convert a signal from ELN to LSF, first the signal has to be converted from ELN to TDF and then from TDF to LSF. The reverse is also true.

SystemC-AMS extensions also support time-domain and frequency-domain simulation. In time-domain simulation, time-domain behavior of the overall system composed of different SystemC-AMS MoCs and probably the discrete-event domain are described. On the other hand, frequency-domain simulation is applied on the cases where the analysis computes the small-signal frequency-domain behavior of the overall system (Barnasconi, *et al.*, 2010).

Again, the use of SystemC-AMS with its MoCs - enables the developer to model and simulate any architecture at all levels. At the implementation level where the developer has variety of choices on the implementation options to make, modeling and simulation of any architecture can be made (Grimm, *et al.*, 2008).

3.4.2 The limitations of SystemC-AMS

SystemC-AMS like any other system has some limitations in system modeling and simulation. The TDF MoC has restrictions caused by its fixed time step mechanism. The MoC has fixed and constant time steps that cannot be changed dynamically. This limitation does not allow developers to easily model systems such as Voltage Controlled Oscillators (VCOs), clock recovery circuit's etc., in which activation periods or frequencies change dynamically (Barnasconi, *et al.*, 2011). Perhaps some of these limitations may be overcome in the future.

To recapitulate, the Model-Simulate-Refine-Synthesis (MSRS) Methodology developed in this work builds on SystemC-AMS to support modeling, architectural exploration and simulation.

Chapters four and five present case studies of two E-AMS systems that have been modeled and simulated using the MSRS methodology to verify and validate the methodology respectively.

CHAPTER FOUR

CASE STUDY 1: OSCILLOSCOPE

4.1 Introduction

This chapter presents the modeling and simulation of an oscilloscope as a case study. The oscilloscope is modeled and simulated following the MSRS methodology. For each level of the methodology, the results are presented. This case study is chosen to verify the methodology.

4.2 The Oscilloscope Functional Model

At the functional level, the main objective is to ascertain the functionality of the oscilloscope. The oscilloscope is presented as a block diagram in Figure 4.1.

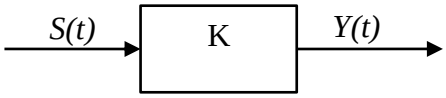


Figure 4.1: Oscilloscope Block Diagram

The block diagram is a black box whose contents are of no interest at this level. The K presented on the black box indicates some processing (that may include attenuation, amplification and/or conversion from analogue to digital) taking place to generate the output *Y(t)* given the input *S(t).* The output can be either analog or digital signals depending on the kind of oscilloscope modeled. In this work, digital oscilloscope is modeled and simulated because digital oscilloscopes are mostly used today. Due to this choice, the block diagram in Figure 4.1 is synthesized to generate a more detailed block diagram presented in Figure 4.2.

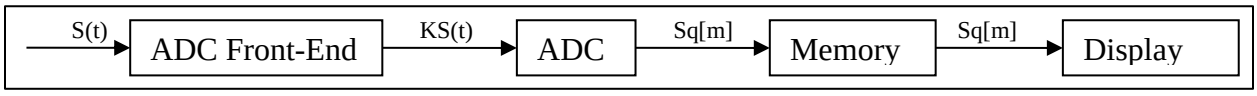


Figure 4.2: Detailed Oscilloscope Block Diagram

The block diagram presented in Figure 4.2 shows the main components interconnected and also the signal flow. The signal flows from the input to the memory. The analogue signal, *S(t)*, is connected to the Analog-to-Digital-Converter (ADC) front-end for scaling purposes which may involve attenuation or amplification of the signal. Once the signal is in the required range of the ADC, it is sampled and converted into digital values, $S_q[m]$, which is stored in a memory. The digital values in the memory are displayed on a screen to show the equivalent signal. The detailed block diagram in Figure 4.2 is then synthesized to generate the dataflow model for the oscilloscope presented in Figure 4.3 though the memory and display components found in Figure 4.2 are not captured in Figure 4.3 because the signal vales after the ADC remain the same as they go through the memory and the display.

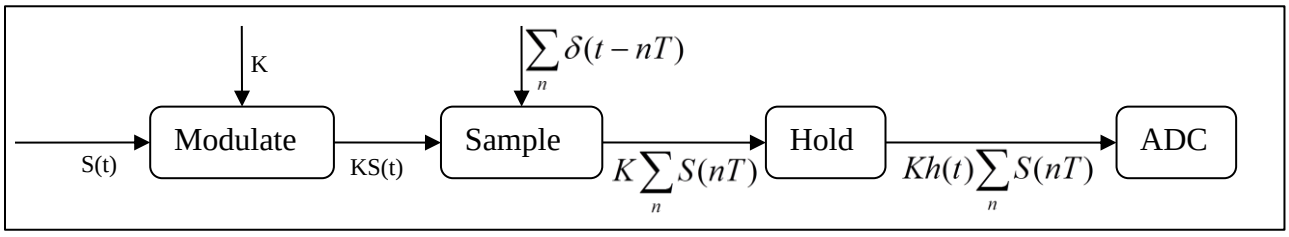


Figure 4.3: Oscilloscope Dataflow Model

The dataflow model presented in Figure 4.3 is converted to a simulation model. In the conversion, the processing elements translate to clusters presented by rectangular boxes. The connections translate to signals and ports as guided by SystemC-AMS (Barnasconi, *et al.*, 2011). They are presented by arrows as shown in Figure 4.4. The simulation model has a test-bench, which comprises of a signal conditioner (K-source), test signal source (Signal source), sampling clock (Pulses) and a monitor. The signal source generates the simulated test-signal, signal conditioner amplifies or attenuates the simulated test-signal, sampling clock determines the sampling of the oscilloscope and the monitor is used to track the signals at every stage of interest. The test-bench and the Device Under Test (DUT) are presented in Figure 4.4.

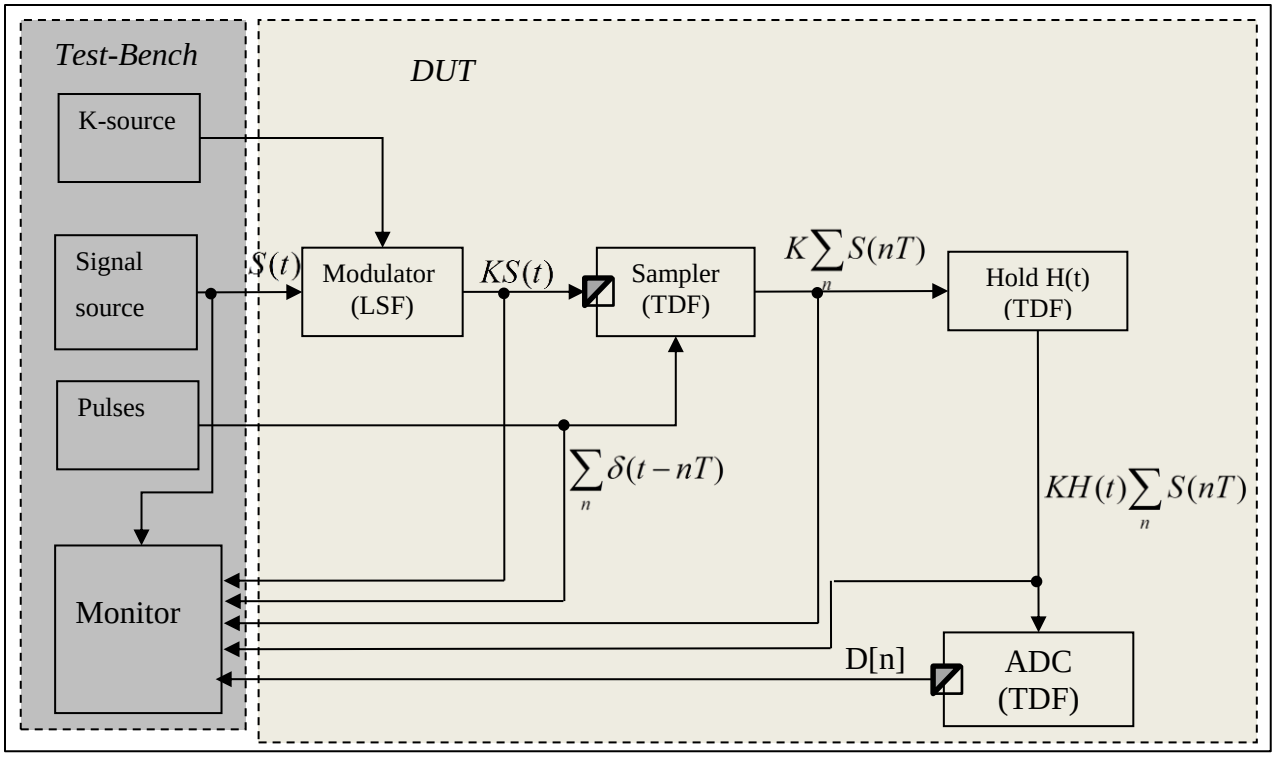


Figure 4.4: Oscilloscope Simulation Functional model with Test-Bench

The signal modulator, which amplifies or attenuates the signals, operates through multiplying the signal by a factor. The multiplication of a signal by a factor is directly supported by LSF model. Due to this fact, the signal modulator (whose code is presented in Appendix A.1) is modeled using the LSF MoC. The sampler samples the signal at given time intervals depending on the sampling clock. The sampling process brings in the idea of timed data and therefore, the sampler is modeled using the TDF MoC. Since the LSF output signal from the modulator serves as the input signal to the sampler (whose code is presented in Appendix A.2) modeled using the TDF MoC, then signal conversion from LSF to TDF is required. SystemC-AMS provides an LSF to TDF converter model for this purpose.

The hold module is used to hold the sampled values until the ADC conversion is complete. Since the sampled values are in TDF form, the hold module is modeled using TDF MoC. The code for the hold module is presented in Appendix A.3. Similarly, the ADC is modeled using TDF MoC since TDF directly supports reading/writing of values from/to ports.

The analog TDF value from the hold module is a decimal value while the digital value generated by the ADC is a binary value. Therefore, the conversion process used in the ADC (whose code is in appendix A.4) is the decimal-to-binary base conversion. The oscilloscope ADC modeled at this level is 8-bit and therefore, the input values range from 0 to 255. This means that the ADC introduces a quantization error (Gray, 2006).

The analog-to-digital value conversion at the functional model stage is implemented using "for-loop" as shown in the code section presented in Figure 4.5.

```
1     samp = (anlgsigin.read());
2     intsampval = (int) (samp);
3     for(i=0; i<8; i++)
      {
4           r = intsampval % 2;
5           dev = intsampval / 2;
6           adcval[i] = r;
7           intsampval = dev;
      }
8     adcout.write(adcval);
```

Figure 4.5: Analog-to-digital value conversion

The code presented in Figure 4.5 shows code statements used in the conversion of the analog values to digital values. Line 1 of the code reads the analog value and stores it in a variable, "*samp*". The analog value is divided by 2 and its remainder stored in a temporary variable, "*r"*, as given in code line 4. The statement in Line 5 divides the sampled value by 2 and stores the result in the variable *"dev"* for further conversion. The statement in line 6 stores the remainder after integer division presented in line 4 in an array *"adcval[]"*. Line 7 transfers the value sored in *"dev"* to the variable *"intsampval"* for further conversion. Line 8 writes the binary value to the output terminal of the signal generator module. This process is repeated until the simulation time is over. The binary values are used to generate the oscilloscope signal waveforms.

The oscilloscope functional model is tested using a 10 KHz sine wave analogue signal. The 10 KHz signal has been selected randomly just to test the functionality of the system with an assumption that it can work with other frequencies as it will be shown in section 4.3. The results

of the oscilloscope functional model are presented in Figure 4.6. In the figure, the signal is monitored at different stages. The S(t) is the analogue signal generated by the signal source and connected to the gain module, KS(t) is the analogue signal after the gain and connected to the sampler. KS(nt) is the sampled analogue signal connected to the Hold, KH(t)S(nT) is the sampled analogue signal after the Hold, which is connected to the ADC, and D(n) [7:0] is the digital signal after the ADC. The oscilloscope modules (Modulator, Sampler, Hold and ADC) are wired together on a test bench whose code is presented in Appendix A.5. The functional model of the oscilloscope is simulated and the results presented in Figure 4.6.

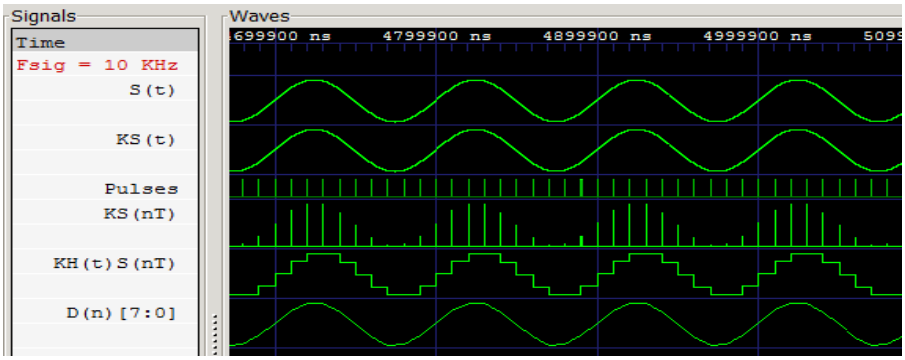


Figure 4.6: Oscilloscope Functional Model Waveforms

Figure 4.6 presents the signal waveforms at different stages. The analogue signal, S(t), is the analog signal connected to the oscilloscope. The modulated signal, KS(t), is of the same shape as the input signal S(t). The amplitude is also the same because the modulating factor, K, is set at this stage to 1. The sampled signal, KS(nT), shows the sampled points of the analogue signal. From the Figure 4.6, it is evident that the sampling is synchronized to the clock pulses (Pulses). The sampled signal is connected to the hold module. The hold signal, KH(t)S(nT), is shown as a staircase waveform, which shows the hold effect on the signal. The hold signal is connected to the ADC for analogue-to-digital conversion. The binary signal, D[n] from the ADC is presented graphically as is typical of an oscilloscope.

As can be seen in Figure 4.6 the input signal S(t) waveform is the similar as the output signal D[n] waveform. Also the waveforms at the various stages concur with the expected waveforms at those stages. This provides strong confidence in the simulation methodology for this stage.

Since the main objective of an oscilloscpe is to sample and reproduce the connected analogue signal, then the results in Figure 4.6 show that the objectives of ascertaing the sampling and reconstruction functionalities of the oscilloscope have been met since the signal sampling is giving credible results. Since the functionality of the oscilloscope is verrified, then the functional model can be synthesized further by considering the non-functional requirements to generate the non-functional model.

4.3 The Oscilloscope Non-Functional Model

Among other non-functional requirements, include resolution, amplitude range and sampling frequency. It is expected that resolution depends on the bit width of the ADC and amplitude range depends on modulation (amplification and/or attenuation). Therefore, the modeling and simulation objectives at this level are to determine the effect of ADC bit width on the resolution, the effect of signal modulation on amplitude range and demonstrate an automated gain control. To achieve the first objective, bit truncation is done. The least significant bits are truncated to generate the same signal but of different bit widths. For the case of signal modulation, signal amplification and attenuation modules are required. Nyquist's theorem states that "the sampling frequency dependents on the signal frequency". Thus, the signal frequency is first determined and then used to generate the sampling frequency. All these features have been used to synthesize the control unit. This unit also synchronizes and monitors the components of the oscilloscope. The oscilloscope non-functional model is presented in Figure 4.7.

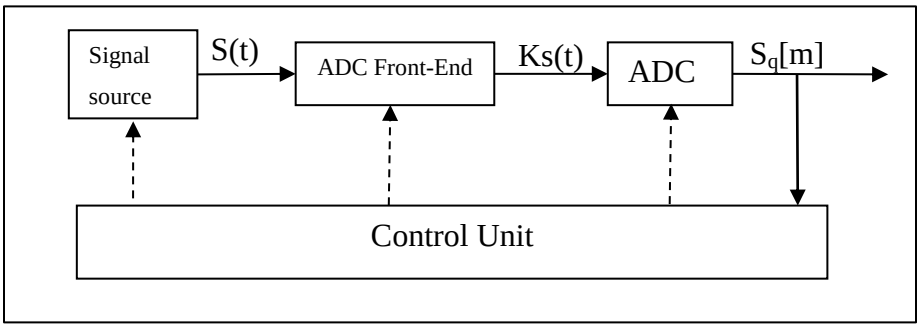
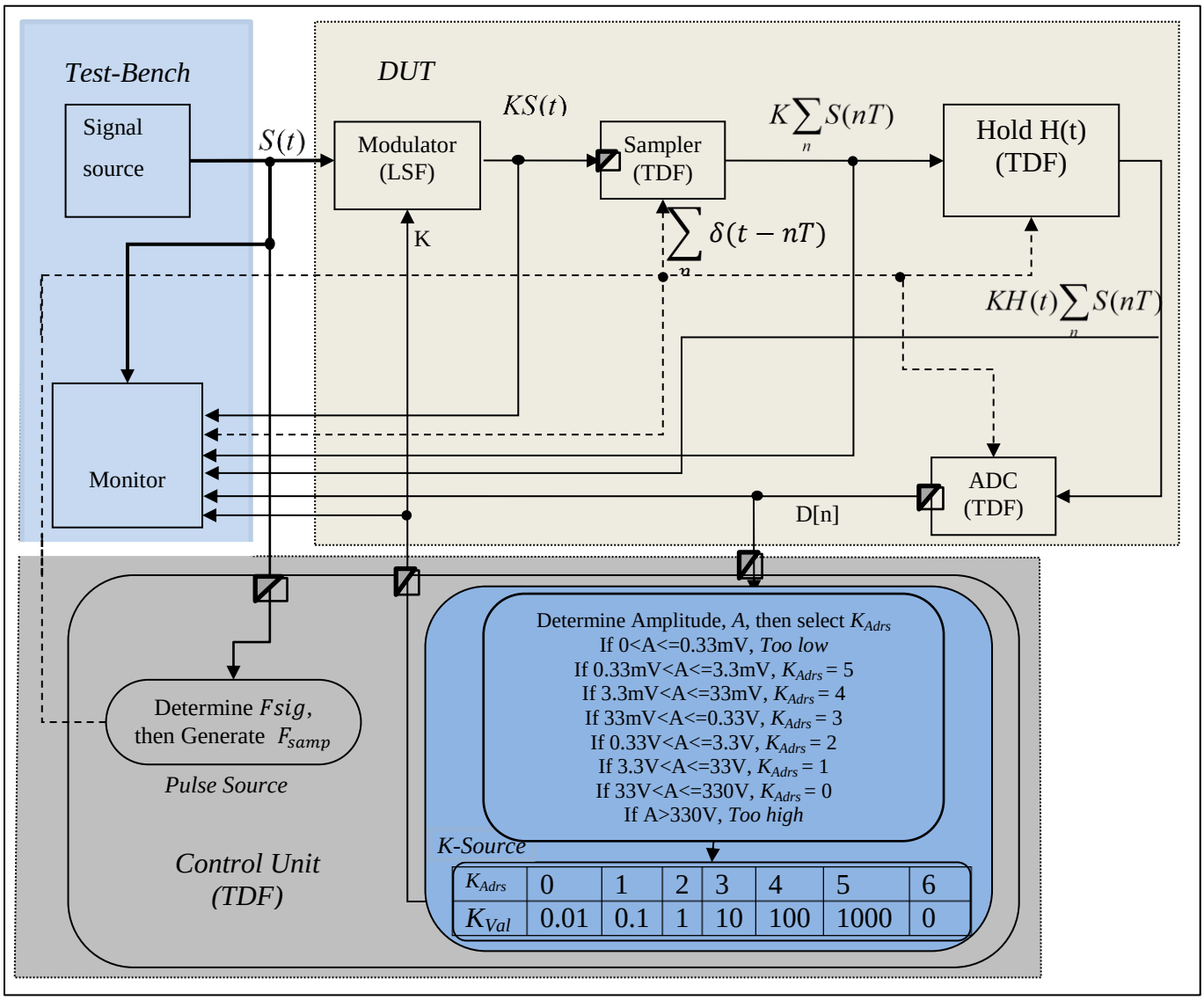


Figure 4.7: Oscilloscope Non-Functional Block Diagram

The oscilloscope non-functional block diagram is synthesized to generate the oscilloscope non-functional simulation model presented in Figure 4.8.



Key: The solid lines depict data flows. The dotted lines depict control signals

Figure 4.8: Oscilloscope Non-Functional Simulation Diagram

The oscilloscope non-functional simulation diagram presented in Figure 4.8 shows how the simulation takes place. It also shows different components of the oscilloscope and how they are controlled by the control unit. The control unit (whose code is in appendix A.6) has two

modules, one that determines the signal frequency and another that sets the sampling clock frequency. The relationship between the signal frequency and the sampling frequency is determined by the number of samples required in one signal cycle. In this case, 20 samples are taken since they give more realistic signals as compared to fewer samples. Equation 4.1 gives the relationship.

$$F_{samp} = 20 \times F_{sig} \tag{4.1}$$

Where: F_{samp} is the sampling frequency and F_{sig} is the signal frequency

The control unit has a module that determines the signal amplitude and chooses the correct amplification factor, K. The K values are stored in a Look Up Table (LUT). From the Figure 4.8, once the signal amplitude range is determined, the LUT address that contains the correct amplifying or attenuating factor is determined and in return, the factor is used by the modulator.

The model is simulated and tested using different signals. At first, the amplitude range is determined through fixing the signal frequency to 0 Hz (DC signal) and varying the amplitude to have 0V, 1mV, 3V, 200V and 330V. Also the signal frequency range is determined through fixing the amplitude to 3V (within the normal range, 1V <A<=3V) and varying the frequency by orders of magnitude to have 30KHz, 300KHz, 3MHz, 30MHz, 300MHz and 4 GHz.

To determine the effect of bit resolution, ADC bit widths of 10, 8, 6, 4 and 2 bits but at the same frequency, 30MHz, and amplitude, 3V, are used. The ADC code is presented in Appendix A.7.

Figures 4.9(a), 4.9(b), 4.9(c), 4.9(d) and 4.9(e) show DC (0 Hz) waveforms but of different amplitudes. The frequency is fixed and amplitude varied, to simulate varying input signal amplitudes. The modules in this level are wired together on a test bench presented in Appendix A.8.

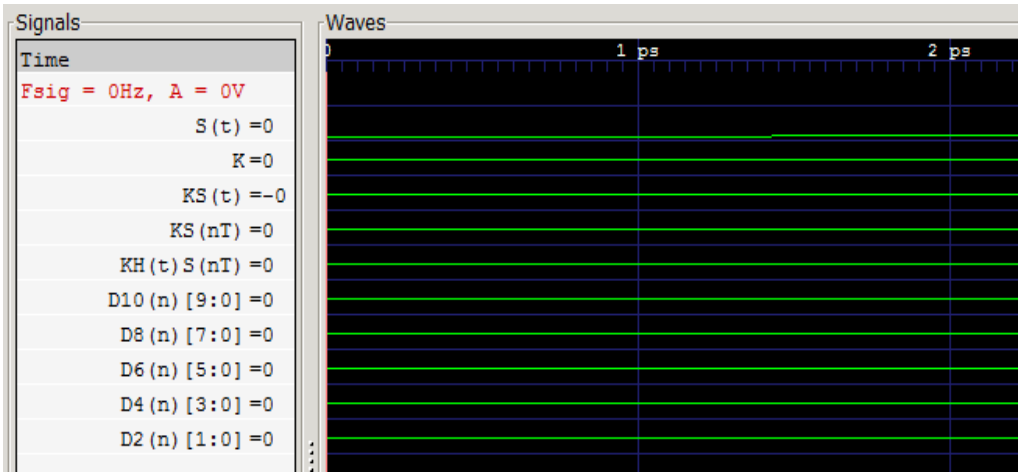


Figure 4.9(a): 0V, 0Hz Analogue and Digital signals for 2, 4, 6, 8 and 10 bits

Figure 4.9(a) present DC signals for a 0 V, 0Hz signal. From the diagram, straight lines of 0 volts are shown. The 0 values are as a result of the 0 volts amplitude of the signal. This is an oversight in relying entirely on simulation and is a potential hazard of pure simulation. It is also observed that the K value is 0. This is consistent with the automatic range setting of the simulated scope since the signal amplitude is 0 which is below the range that he oscilloscope operates.

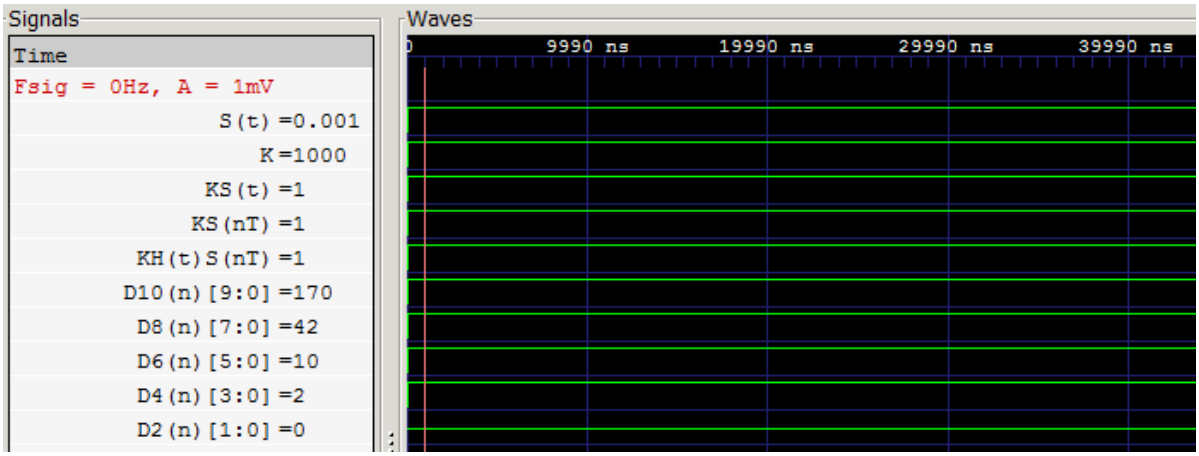


Figure 4.9(b): 1mV, 0Hz Analogue and Digital signals for 2, 4, 6, 8 and 10 bits

The DC signals presented in Figure 4.9(b) are for 1mV DC signal. The signals here are straight lines of 1mV. The values for the digital signals correspond to the equivalent voltages based on the ADC bits in each case. This shows the minimum voltage amplitude that can be amplified.

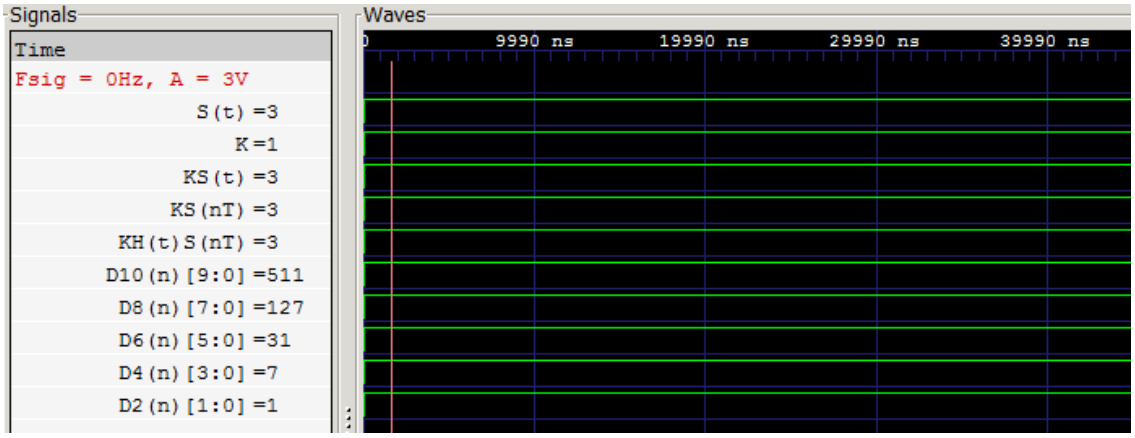


Figure 4.9(c): 3 V, 0Hz Analogue and Digital signals for 2, 4, 6, 8 and 10 bits

In the case of Figure 4.9(c), the signals are for a 3 V DC signal. They are also straight lines but of 3 volts, which, is within the normal range that do not required any amplification or attenuation. This signal is used to test the working of the oscilloscope at the normal signal range where attenuation or amplification is not required.

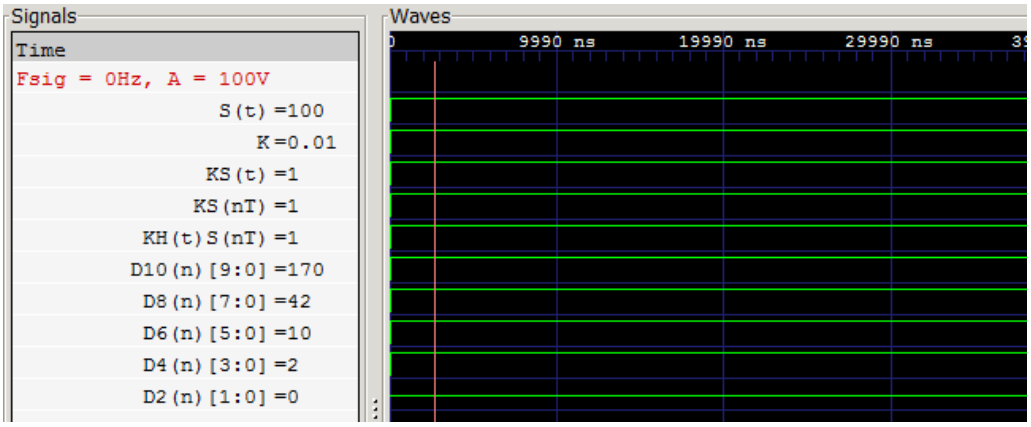


Figure 4.9(d): 100V, 0 Hz Analogue and Digital signals for 2, 4, 6, 8 and 10 bits

Figure 4.9(d) shows a 100 V DC signals. It is shown that there is an attenuation by 100 since the K value is 0.01. This attenuates the signal amplitude from 100 V to 1 V. At the same time, just like in the case of Figure 4.9 (b) and (c), the digital signal values correspond to their equivalent values according to the ADC bits.

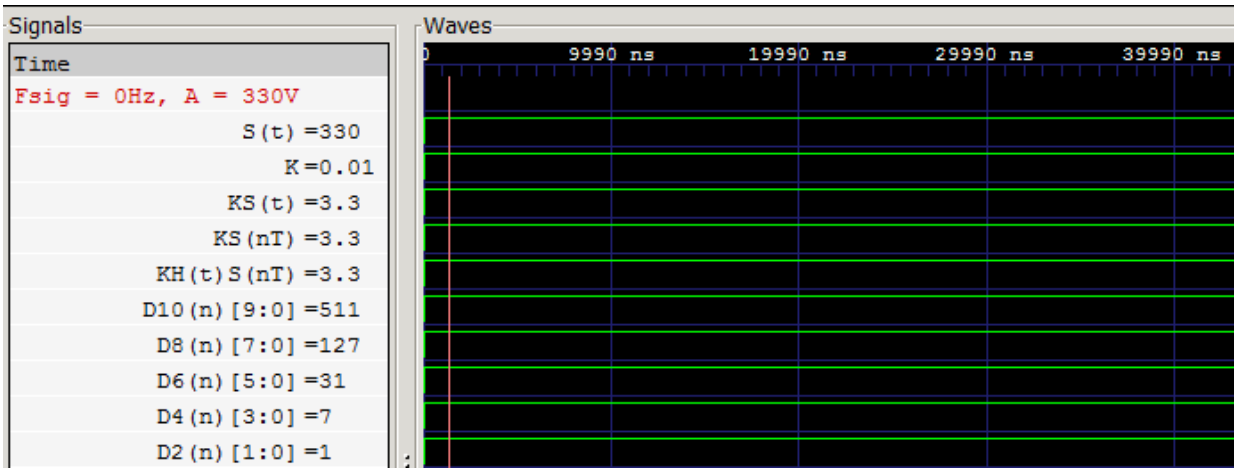


Figure 4.9(e): 330V, 0Hz Analogue and Digital signals for 2, 4, 6, 8 and 10 bits

In the Figure 4.9(e), a 330 V DC signal is presented. The figure shows attenuation by 100 done to the maximum signal amplitude in this case. Any signal, whose amplitude is above the 330 V, is treated as out of range.

Figures 4.9(f), 4.9(g), 4.9(h), 4.9(i), 4.9(j) and 4.9(k) show waveforms for 3 V amplitude and different frequency signals. The fixed amplitude of 3 V is chosen because it is within the normal range but any other amplitude within 0.33mV and 330V could have been chosen since the oscilloscope can operate within that range. Different signal frequencies as presented in the figures are used to simulate the oscilloscope. In this simulation the effect of resolution on reconstruction of the signal and sampling frequency are explored and an automated gain control generated successfully.

The different resolutions are achieved by having ADCs of different bits, 2, 4, 6, 8 and 10, sampling the same analogue signal. The relationship between the resolution and the bit-width is presented in Equation 4.2 and summarized in Table 4.1.

$$Resolution, r = \frac{Amplitude, A}{N} = \frac{Amplitude, A}{2^n - 1} \quad (4.2)$$

Where *n* is the ADC bit-width

Table 4.1: The effect of ADC bit-width on Resolution

Digital signal	ADC Bits, n	$N = 2^n-1$	Resolution, r = A/N
D10(n)[9:0]	10	1023	2.93mV
D8(n)[7:0]	8	255	11.76mV
D6(n)[5:0]	6	63	47.62mV
D4(n)[3:0]	4	15	200mV
D2(n)[1:0]	2	3	1000mV

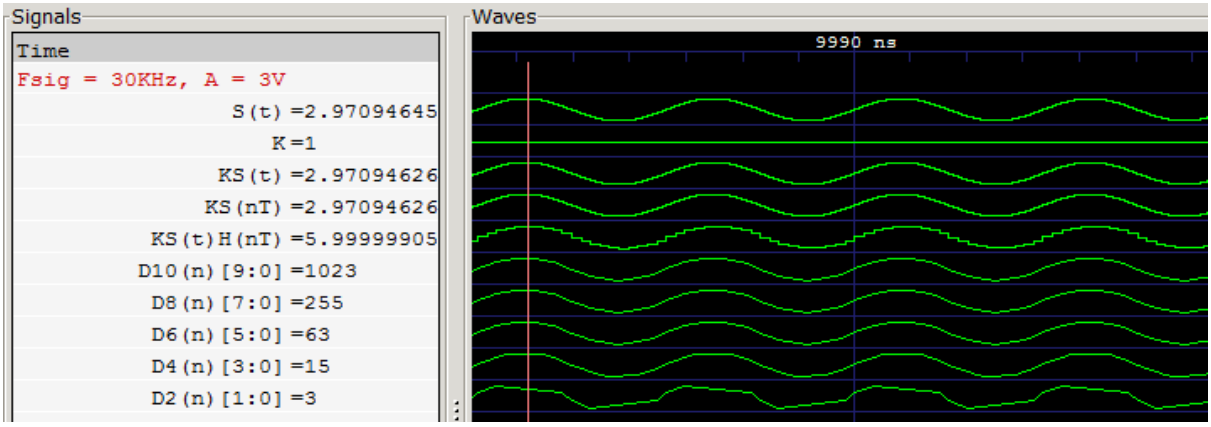


Figure 4.9(f): 3 V, 30KHz Analogue and Digital signals for 2, 4, 6, 8 and 10 bits

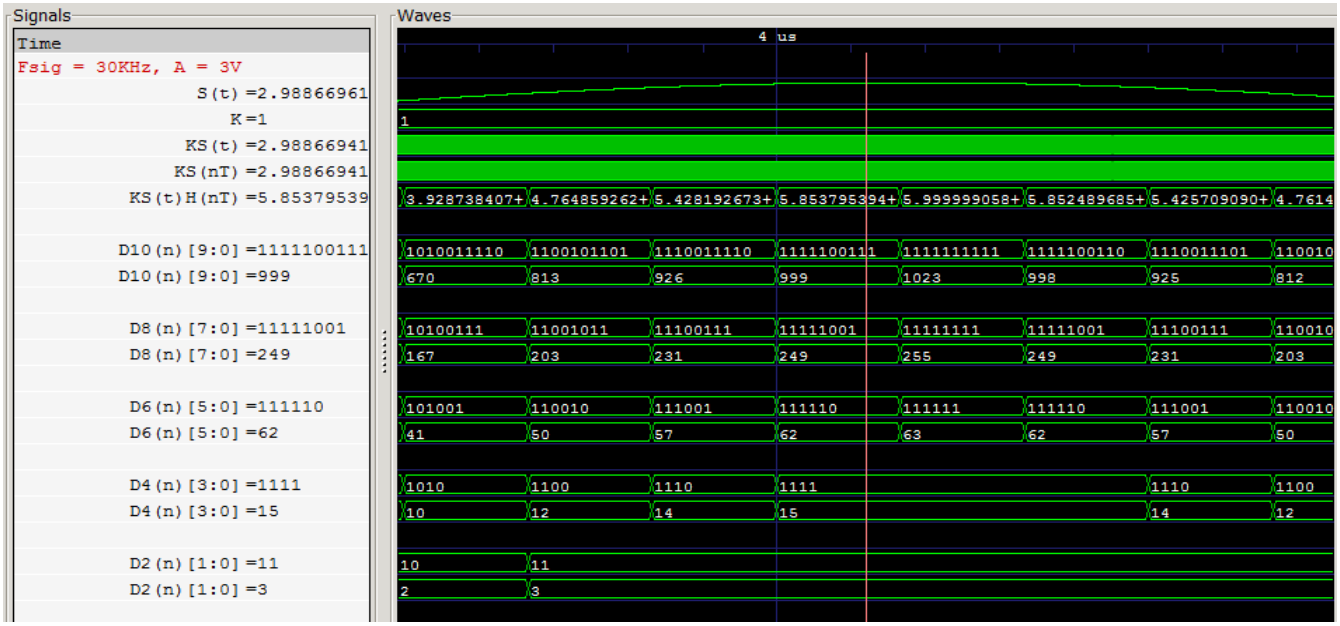


Figure 4.9(g): 3 V, 30KHz Analogue and Digital time diagrams for 2, 4, 6, 8 and 10 bits

The time diagrams presented in Figure 4.9 (g) are for the waveforms presented in Figure 4.9 (f) and show how the same analogue signal is sampled at different ADC bits. For each digital signal, the signal is presented in binary and decimal forms for comparison purposes.

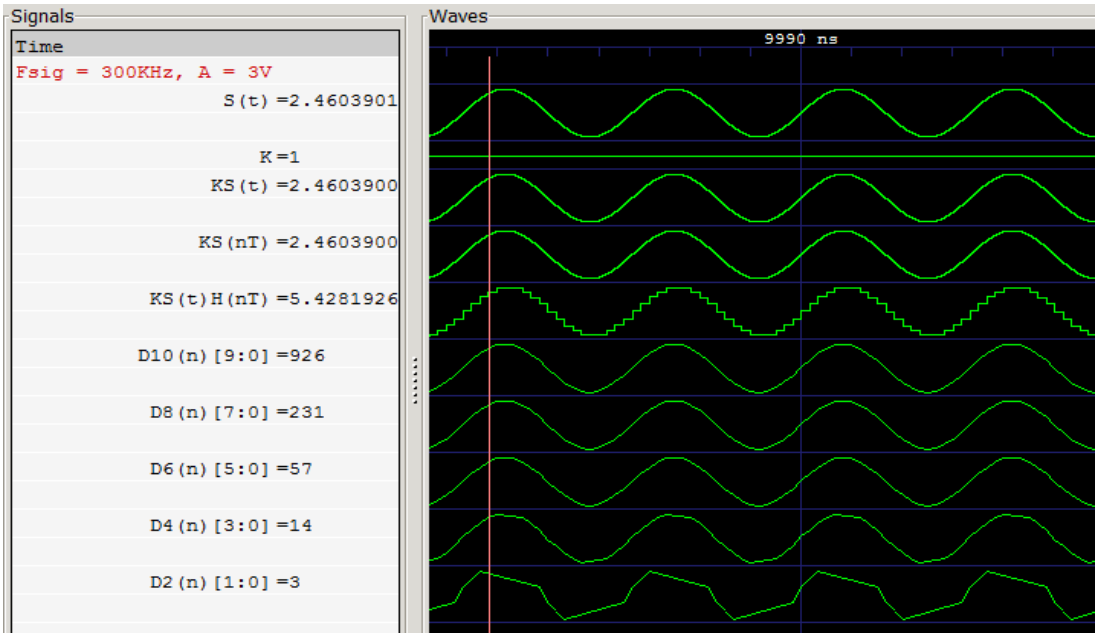


Figure 4.9(h): 3 V, 300KHz Analogue and Digital signals for 2, 4, 6, 8 and 10 bits

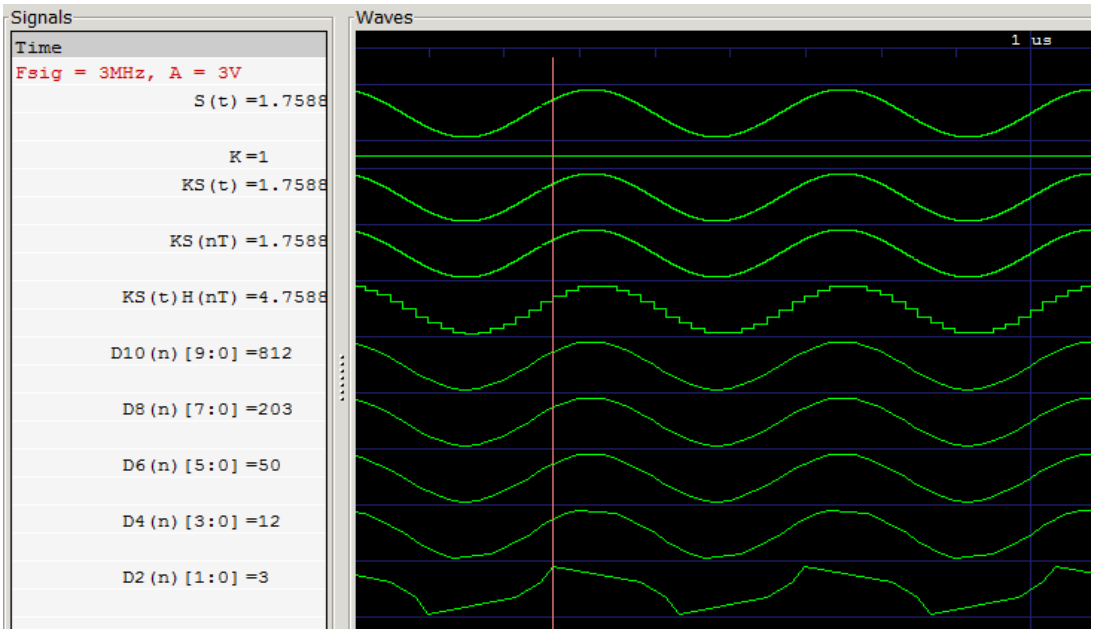


Figure 4.9(i): 3 V, 3 MHz Analogue and Digital signals for 2, 4, 6, 8 and 10 bits

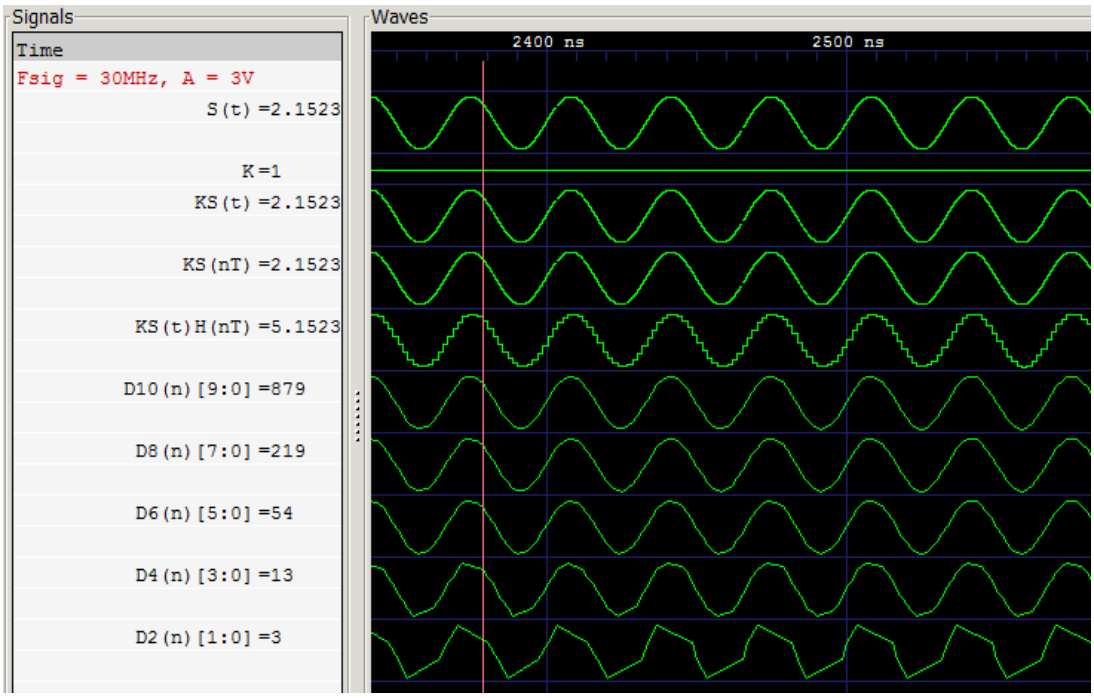


Figure 4.9(j): 3 V, 30 MHz Analogue and Digital signals for 2, 4, 6, 8 and 10 bits

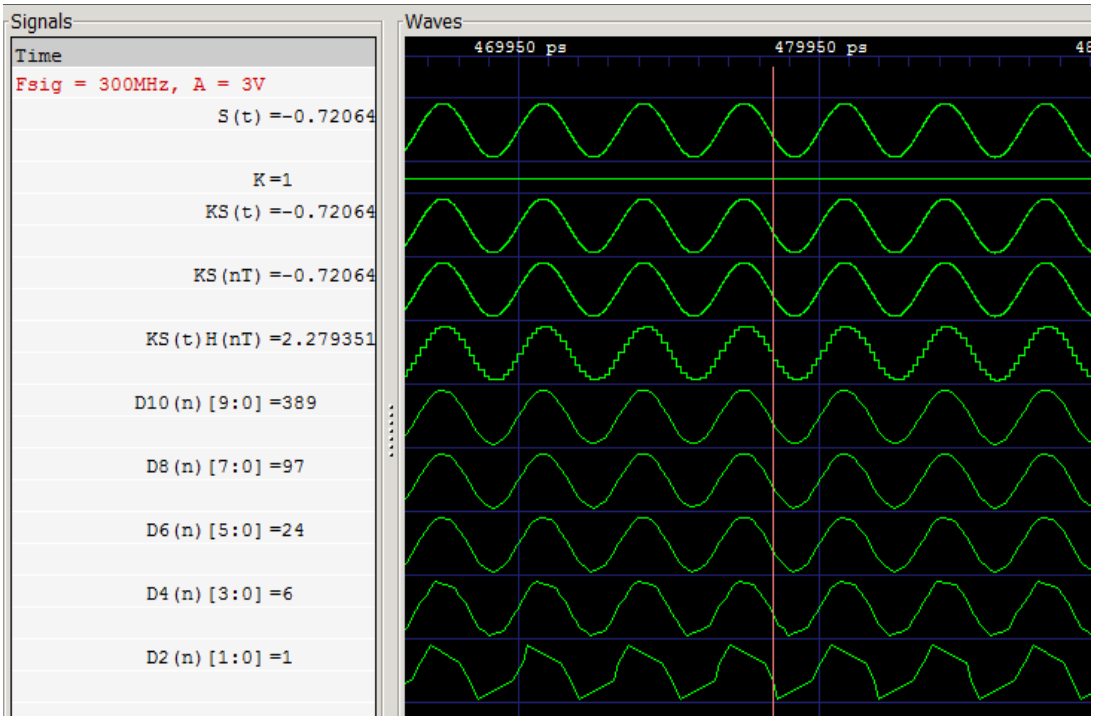


Figure 4.9(k): 3 V, 300 MHz Analogue and Digital signals for 2, 4, 6, 8 and 10 bits

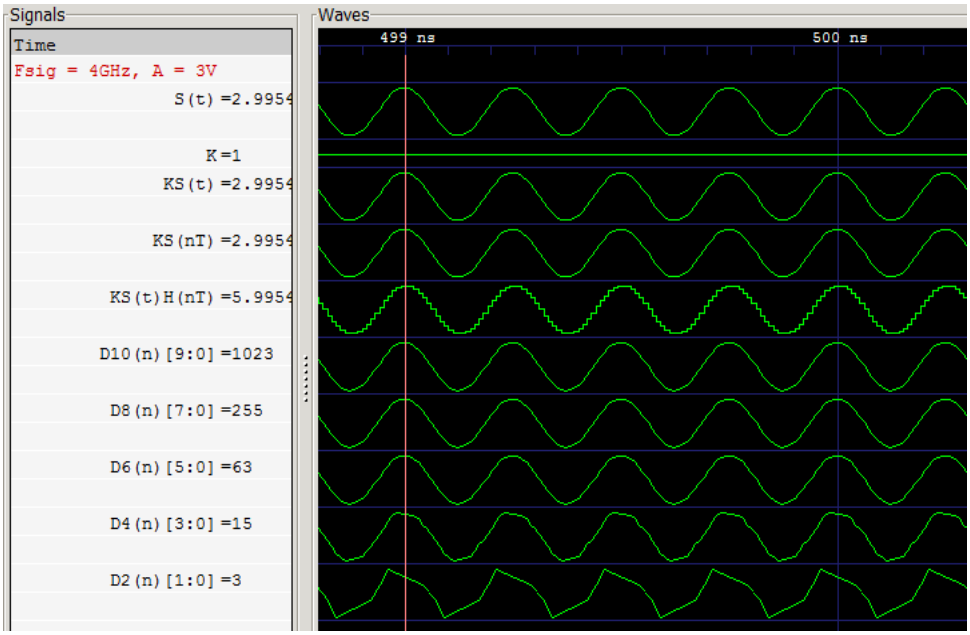


Figure 4.9(l): 3 V, 4 GHz Analogue and Digital signals for 2, 4, 6, 8 and 10 bits

The waveforms presented in the Figures 4.9(f) to 4.9(l) show sampling and reconstruction of the oscilloscope signals. This shows that the modeled oscilloscope can sample signals of different frequencies and give faithful reconstruction as long as the resolution is appropriate. The Figure 4.9(k) shows that oscilloscopes operating at frequencies as high as 4 GHz can be simulated. At 20 samples per signal cycle, this will require a sampling frequency of 80 GHz. This emphasizes the importance of finding the minimum number of samples per signal cycle that are adequate for the reconstruction of the signal.

Comparing the digital signals from the Figures 4.9(a) to 4.9(k), it is found that different bits are used to present signals of the same frequency. From the digital signals, especially in Figures 4.9(f) to 4.9(l), it is evident that signals of higher bits are more faithfully reconstructed than signals of low bits. For each frequency, comparing the 10 bit digital signal and the 2 bit digital signal, the 10 bit signals have better reconstruction results in relation to the 2 bit counterparts. This can be explained by the fact that reducing the ADC bits increases quantization noise in proportion to the number of bits. This has the effect of making faithful reconstruction more difficult.

The reconstructed waveforms for 8 bit and 10 bit are adequately representative of the input signal. Therefore, for the optimal architectural model, the 8 bit resolution is chosen, as it is more efficient in storage than 10 bit resolution.

4.4 The Oscilloscope Implementation Model

The oscilloscope implementation model is a synthesis of the optimal architectural model and generic component libraries. The LSF and TDF models used in the oscilloscope functional model presented in Figure 4.4 are converted into ELN modules used at the implementation stage of the system. This is one of the main features that make SystemC-AMS attractive for this task. This generates the implementation model of the oscilloscope. The implementation model generated is simulated and the results compared with the results from the non-functional model. Therefore, the objectives at this level are; synthesis of the implementation model from the optimal architectural model and generic components libraries and the evaluation of the implementation model with reference to the optimal architectural model. In addition, the implementation model demonstrates automation frequency selection. Figure 4.10 shows the implementation model.

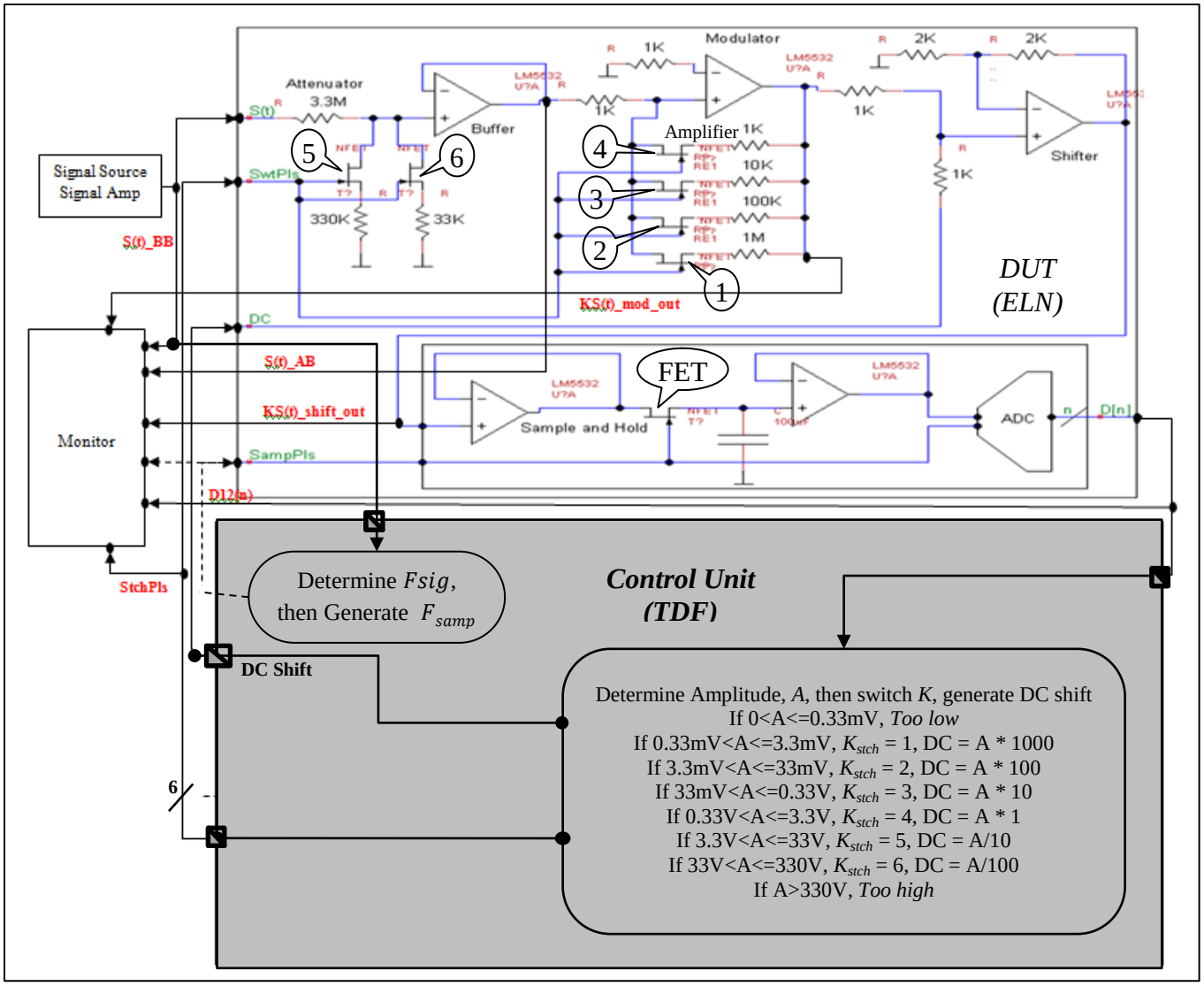


Figure 4.10: The Oscilloscope Implementation Model

The interconnections of the modules forming the cluster simulated are used to generate its prototype. The prototype components consist of the modulator (the attenuator/buffer and the amplifier), signal DC shift (shifter), sample and hold (Sample and Hold) and the ADC (whose codes are in Appendix A.9, A.10, A.11, A.12 and A.13 respectively). The transistors numbered 1 to 6 act as the switches for the amplifying and attenuating circuits of the DUT. When transistors 5 and 6 are switched, they attenuate the signal by a factor of 10 and 100 respectively. Switching transistors 1, 2, 3 or 4, amplifies the signal by a factor of 1000, 100, 10 and 1 respectively. The

signal shift circuit is used to shift the signal by some DC volts generated by the control unit. DC shift is usually required because, components such as ADCs generally work with only positive voltages. The sample and hold circuit is generated using operational amplifiers, FET transistor and a capacitor. The FET transistor and the capacitor serve as the switch and the sampled value storage respectively. The attenuator/buffer , amplifier, shifter, Sample and Hold and the ADC modules are wired together on a test bench presented in Appendix A.14.

The oscilloscope system developed is simulated using different signals of different frequencies and amplitudes. The signal frequencies and amplitudes chosen at this level are to compare with the waveforms presented in section 4.3. In addition, the output ADC digital signal bit width is set to 8 bits. The modeling and simulation at this level is made for verification of the results.

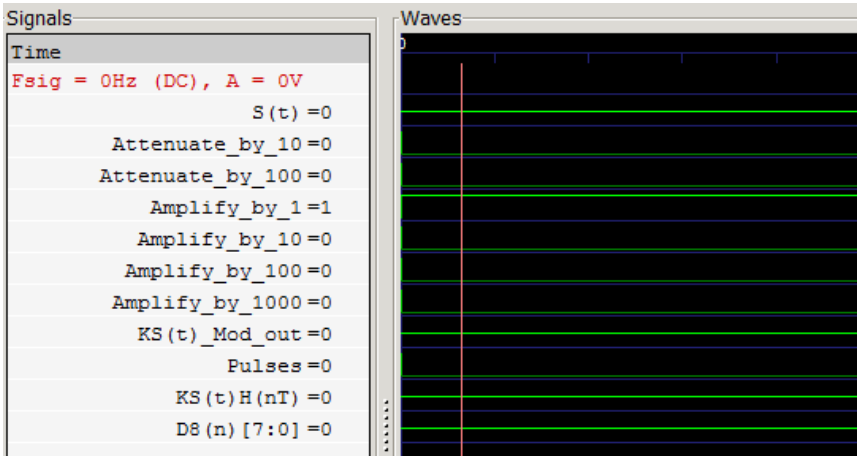


Figure 4.11(a): 0 V, Analogue and 8-bit Digital signals at 0Hz

Figure 4.11(a) presents a 0 volts DC signal. The signals are presented as straight lines because they are DC signals. Again, from the figure, all the signals have zero values because the amplitude of the analogue signal is 0 volts. Comparing the 0 volts DC waveforms in Figure 4.9(a) (for the non-functional model) with the waveforms presented in Figure 4.11(a) (for the implementation model), it is found that they are the same.

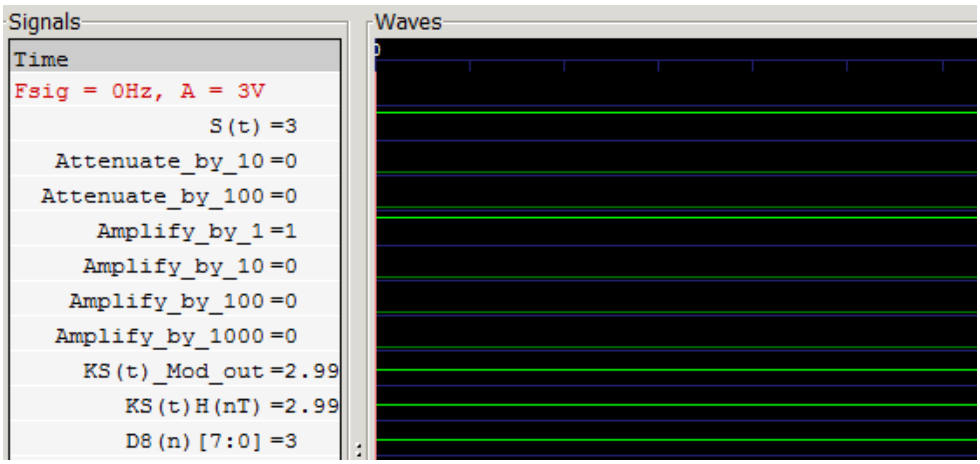


Figure 4.11(b): 3 V, Analogue and 8-bit Digital signals at 0 Hz

The waveforms presented in Figure 4.11(b) are for a 3 volts DC signal. The straight lines are because of the DC analogue signal connected. The amplitude of the DC signal is found to be 3 volts as shown by the values of the signals. Comparing this figure with the waveforms in Figure 4.9(c) from the non-functional model, it is found that they are the same.

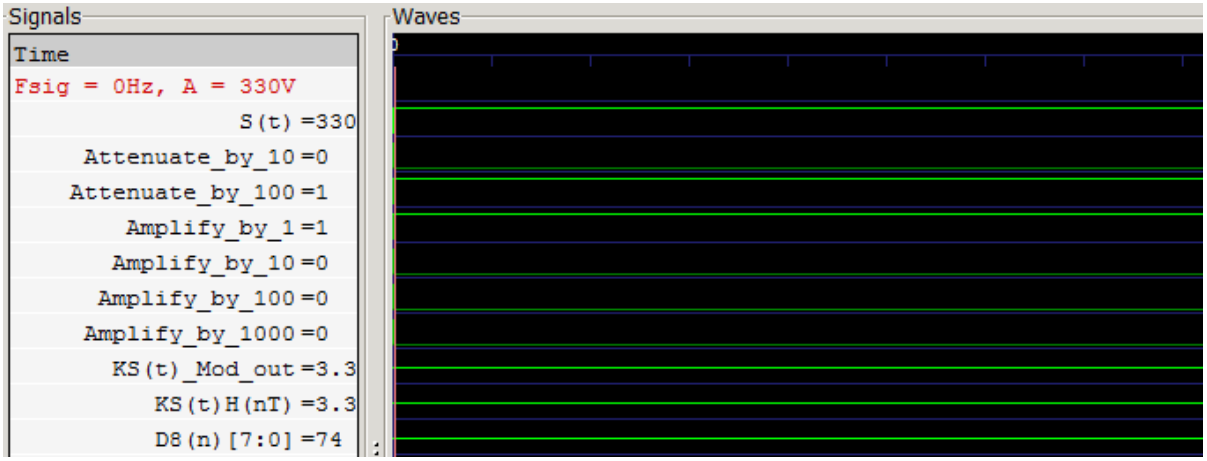


Figure 4.11(c): 330 V, Analogue and 8-bit Digital signals at 0Hz

A 330V DC signal waveforms is presented in Figure 4.11(c). As the other DC waveforms presented, the waveforms in this figure are straight lines. Associating the DC signals in this figure with the DC signals in Figure 4.9(e) of the non-functional model, it is found that they are the same.

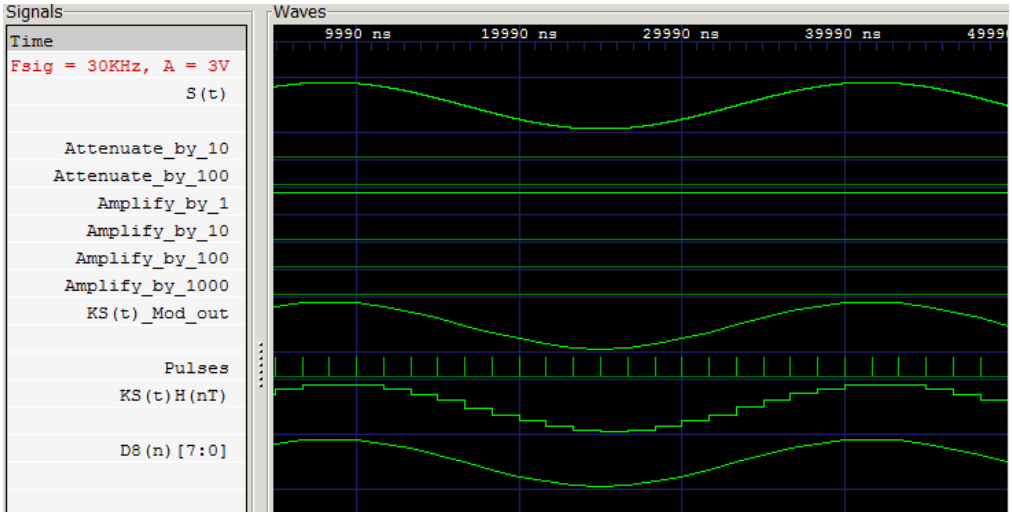


Figure 4.11(d): 3 V, Analogue and 8-bit Digital signals at 30 KHz

Figure 4.11(d) presents a 3 volts 30 KHz signal. Comparing the waveforms with the waveforms on Figure 4.9(f) of the non-functional model, it is evident that they are similar.

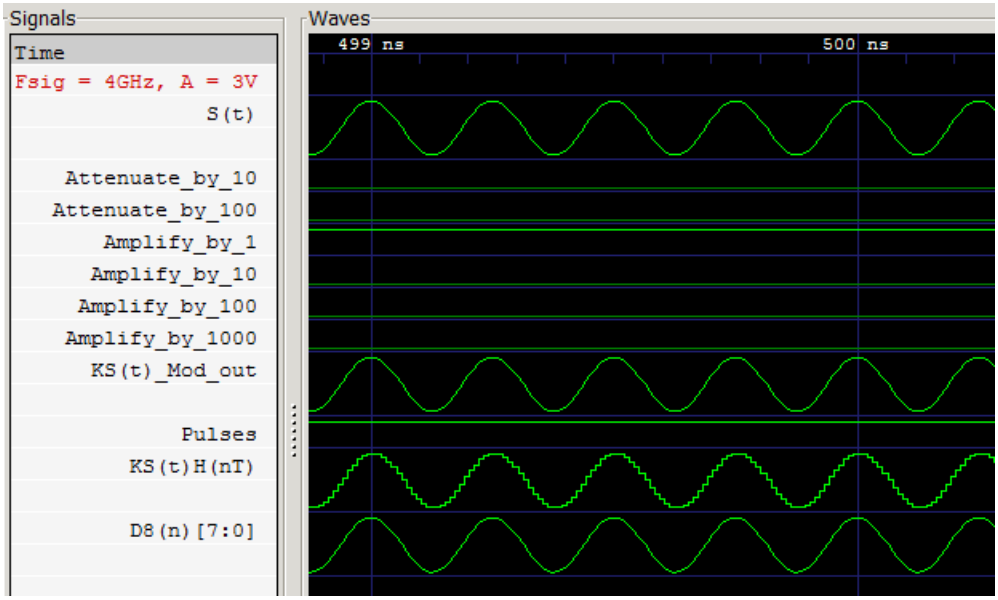


Figure 4.11(e): 3 V, Analogue and 8-bit Digital signals at 4 GHz

As presented from the non-functional model results in Figure 4.9(k), the waveforms presented on Figure 4.11(e) show the same signal. The 4 GHz signal presented in the two figures reflect the

same waveforms, which is evidence that the implementation model simulated gives the same results as the non-functional model.

The waveforms in this section, are the same as their counterparts presented in the non-functional model. This verifies the implementation model with reference to non-functional model. Therefore, the objectives of the implementation model at this level have been achieved.

In summary, the modeling and simulation of a digital oscilloscope has been presented. The modeling and simulation is based on the developed modeling and simulation methodology. The simulation results of the functional model are used to validate the model. The simulation results of the implementation model are used to verify the model against the functional model. Therefore, this makes the methodology developed viable in modeling and simulation of E-AMS systems. Lastly, aware of the apprehension of using simulation for validation, the simulations are compared with a real implementation in the next chapter.

CHAPTER FIVE

CASE STUDY 2: SIGNAL GENERATOR

5.1 Introduction

This chapter presents the modeling and simulation of a signal generator as a case study. The signal generator is modeled and simulated following the MSRS methodology. The simulation results are compared with test results obtained from a similar signal generator developed and implemented in a Fusion FPGA by Muteithia (Muteithia, 2014). This is done to validate the methodology with a real implementation.

5.2 Signal Generator Functional Model

At the functional level, the signal generator is presented using a dataflow diagram as shown in Figure 5.1. The dataflow diagram shows different functional blocks used in generating the functional behavior of the physical system, in this case the signal generator. A signal generator is typically used to generate four different types of analog voltage signals, which include the sine wave, saw tooth wave, triangle wave and square wave as per the stakeholders attributes. The analogue signals are characterized by amplitude, frequency and DC shift values. Therefore, functions are used to manipulate the amplitude, frequency and DC shift values.

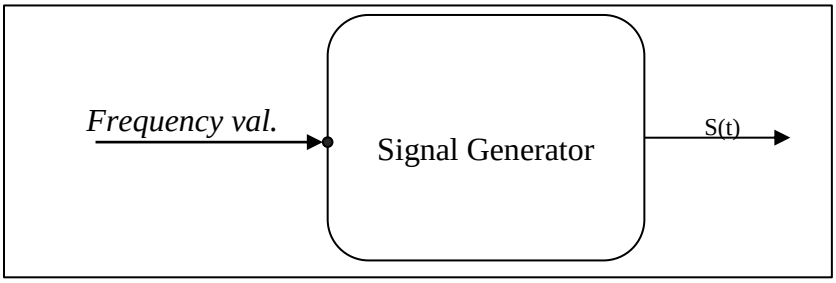


Figure 5.1: Dataflow diagram for the Signal Generator

In Muteithia's work, the starting point at the theoretical background commits *a priori* to a specific architecture. This is considered a disadvantage. With MSRS methodology, no such commitment need to be made at this stage.

The dataflow diagram presented in Figure 5.1 presents the DUT and is connected to a test bench with the driver and the monitor as shown in Figure 5.2. Since Muteithia's signal generator was digital, a digital signal generator is modeled in this work.

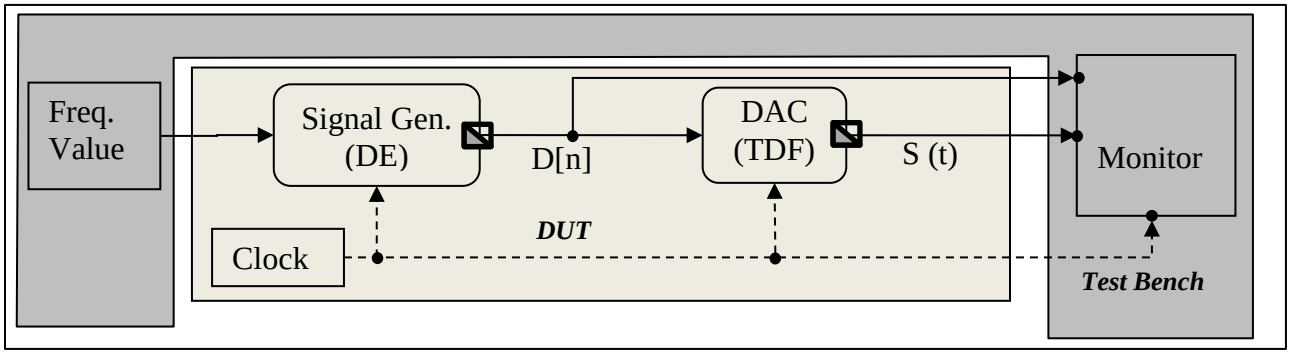


Figure 5.2: Signal Generator with Test Bench

The signal generator module (whose code is in Appendix B.1) is used to generate the digital signals based on input frequency value *f*. the module defines mapping functions that use *f* to generate the required signals. For simplicity, amplitude and DC shift are fixed. Equation 5.1 shows the mapping function used to generate sine wave signal digital values (Lyons, 2012).

$$D_s[\theta] = \sum_n A[\sin(\theta mod_s(n)) \pm k] \quad (5.1)$$

Where:

$\theta = 2 * \pi/s$, sampling interval in radians and s is the number of samples per signal cycle

$s = 1/(f * T_{samp})$, f is the signal frequency and T_{samp} is the sampling period

A is the signal amplitude

k is the DC shift

mod is for modulo operation

$D_s[\theta]$ is the equivalent sine wave digital signal value generated over one cycle

The equation is used to generate an algorithm used to generate the simulation code. The algorithm is presented in Algorithm 5.1.

Step 1: Multiply the product of 2 and π by entered frequency value, f

Step 2: Multiply the product in Step 1 by the instantaneous sampling period,T_{samp}

Step 3: Add the DC shift, k, to the product in step 2

Step 4: Get the equivalent sine wave value of the value obtained in Step 3

Algorithm 5.1: Algorithm to generate single sine wave value

In the case of generating saw-tooth signal wave, a similar mapping equation, Equation 5.2, is used (Lyons, 2012).

$$D_{st}[\theta] = \sum_n A[\theta mod_s(n) \pm k] \quad (5.2)$$

Where:

$D_{st}[\theta]$ is the equivalent saw-tooth wave digital signal value generated

$\theta = 2 * i/s$, sampling interval in radians and s is the number of samples per signal cycle

$s = 1/(f * T_{samp})$, f is the signal frequency and T_{samp} is the sampling period

k is the DC shift

A is the signal amplitude

The equation is used to generate an algorithm presented in Algorithm 5.2 used for the simulation code.

Step 1: Multiply the incrementing integral value,i, by the entered frequency, f

Step 2: Multiply the product in Step 1 by the sampling period,T_{samp}

Step 3: Add the DC shift, k, to the product value in step 2

Step 4: Get the equivalent saw-tooth wave value of the value obtained in Step 3

Algorithm 5.2: Algorithm to generate single saw-tooth signal wave value

Triangular wave signal values are generated using Equation 5.3. This equation is used to generate only the values used in the rising ramp of the signal. The falling ramp is generated through reproducing the values used in the rising ramp but in a reversed way. This means the

values used in the rising ramp can be stored in an array while being produced so that they can be used in the falling ramp (Lyons, 2012).

$$D_{tg}[\theta] = \begin{cases} \sum_n A[\theta mod_s(n) \pm k]\,; for\ mod_s(n) < \frac{s}{2} \\ \sum_n A\left[\left(\theta\frac{s}{2} - \theta\left(mod_s(n) - \frac{s}{2}\right)\right) \pm k\right]; for\ mod_s(n) > \frac{s}{2} \end{cases} \quad (5.3)$$

Where:

$D_{tg}[\theta]$ is the equivalent triangular wave digital signal value generated

$\theta = 2 * i/s$, sampling interval in radians, i is an incrementing integral value and s is the number of samples per signal cycle

$s = 1/(f * T_{samp})$, f is the signal frequency and T_{samp} is the sampling period

k is the DC shift

A is the signal amplitude

The equation is used to generate an algorithm presented in Algorithm 5.3 used to generate the simulation code.

Step 1: Multiply the incrementing integral value, i, by 2
Step 2: Multiply the product in Step 1 by the frequency entered, f
Step 3: Multiply the product in Step 2 by the sampling period, T_{samp}
Step 4: Multiply the value obtained in step 3 by the modulus of n
Step 5: Determine if mod(n) is less than s/2 or not
Step 6: If step 5 is true, add k and output
Step 7: If step 5 is not true, multiply the value of step 3 by s
Step 8: Subtract value of step 4 from value of step 7
Step 9: Add k to value of step 8 and the output

Algorithm 5.3: Algorithm to generate single triangular signal wave value

In the case of generating a square wave signal values, Equation 5.4, is used (Lyons, 2012).

$$D_s[\theta] = \sum_n \alpha_n = \begin{cases} A[\alpha \pm k];\ for\ mod_s(n) < \frac{s}{2} \\ A[-\alpha \pm k]; for\ mod_s(n) > \frac{s}{2} \end{cases} \quad (5.4)$$

Where:

α is a constant

k is the DC shift

A is the signal amplitude

The equation is used to generate an algorithm presented in Algorithm 5.4 used to generate the simulation code.

Step 1: Determine if mod(n) is less than s/2

Step 2: If step 1 is true, Add DC shift, k, to the constant, α

Step 3: If step 1 is not true, Add - α to the DC shift, k

Step 4: Multiply the value in step 2 or step 3 by the signal gain, A then output

Algorithm 5.4: Algorithm to generate single square wave signal value

The Algorithms 5.1, 5.2, 5.3 and 5.4 shows the steps required to generate different signal waves. The steps in each algorithm can be used as a rough guide to the number of instructions required for each signal wave. Algorithm 5.1 shows the four steps required to generate a single value of the sine wave signal hence four instructions are required. Similarly, Algorithm 5.2 shows the four steps required to generate a single value of the saw-tooth wave. This leads to four instructions required to generate a single value of the signal. In the same way, Algorithm 5.3 shows nine steps required to generate a single value of the triangular wave. This gives nine instructions required to generate a single value of the triangular wave. In the case of the square wave, Algorithm 5.4 shows the four steps required to generate a single value of the square wave. Consequently, a single value of the square wave requires four instructions to be generated.

Table 5.1: Instructions required per value for the respective signal waves

	Sine wave	Saw-Tooth wave	Triangular wave	Square wave
Required instruction(s) per value	4	4	9	4

The Table 5.1 shows the four signals and their respective instructions required to generate a single value for each signal. Since the four signals are generated by the same signal generator, the instructions required for each value have the same period and frequency. For this reason, the nine instruction case is used to estimate the instruction period and frequency. The results are presented in Table 5.2.

Table 5.2: Signal generator simulation analysis

Signals freq., F_{sig}	**Values per cycle, S_c**	**Sampling freq., F_{samp}**	**Instr. per value, V_i**	**Instruction Freq., $F_{samp} \times V_i$**	**Remarks**
10 Hz	20	200 Hz	9	1800 Hz	Minimum
1MHz	20	20 MHz	9	180 MHz	Normal
5 MHz	20	100MHz	9	900MHz	Maximum

The Table 5.2 shows the simulation values for 10 Hz, 1 MHz and 5 MHz frequency signals. For each signal frequency, nine instructions for each signal value and 20 values for each complete cycle of the signal waveform are required. Based on these values, it is evident from the table that the signal generating module should be sampling at a frequency between 200 Hz and 100 MHz. Sampling at a frequency less than 200 Hz was found to generate large (larger than 1.5 GB) Value Change Dump (VCD) file used to trace the AMS waveforms. The large VCD files are hard to open and sometimes they may fail to open completely (Barnasconi, *et al*., 2010) , hence the choice of the 200 Hz as the minimum sampling frequency. On the other hand, the 100 MHz sampling frequency was decided for the purpose of comparing the signal frequency and the sampling frequency. This is because in simulation, higher sampling frequencies can be achieved (Barnasconi, *et al.*, 2010). Therefore, to generate a 5 MHz signal, the signal-generating module should be operating at the sampling clock speed, 100 MHz. These serve as some performance attributes of the signal generator, hence are presented at the non-functional level.

Once the values are generated, they are used to generate signals through the Digital-to-Analog-Converter (DAC) whose code is presented in Appendix B.2. the Signal Generator and DAC modules are wired together in a test bench whose code is presented in Appendix B.3. The results of the signal generator functional model are presented in Figures 5.3(a), 5.3(b) and 5.3(c) that show sine wave, saw-tooth wave, triangle wave and a square wave. The four waves are of the

same frequency as shown in each figure. Besides, each signal cycle is constructed using 20 samples as this gives more realistic waveforms as compared to the use of fewer samples. Therefore, 20 samples number is taken as the least number of samples required to construct a single waveform.

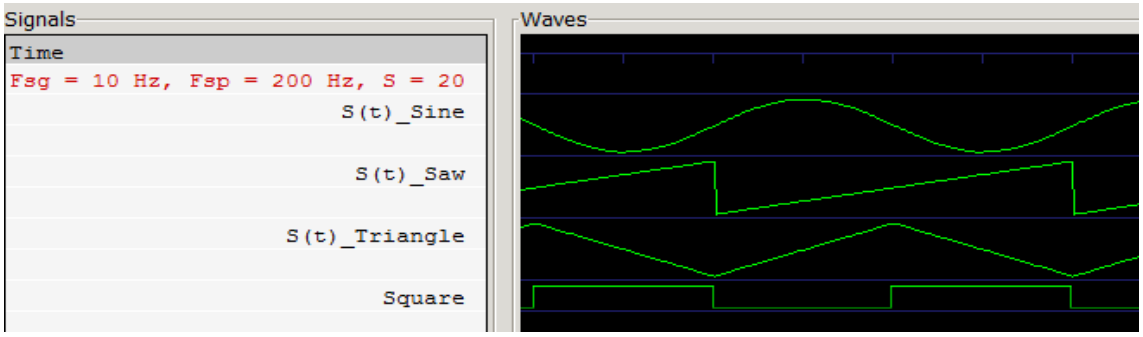


Figure 5.3(a) 10 Hz Signal generator simulation waveform

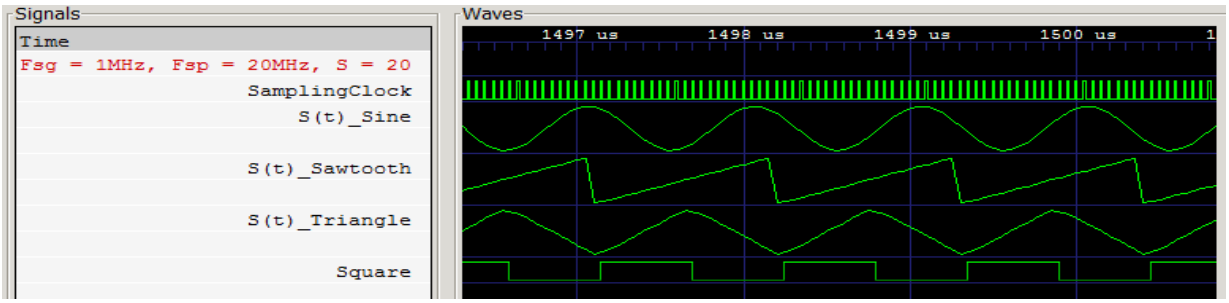


Figure 5.3(b) 1 MHz Signal generator simulation waveforms

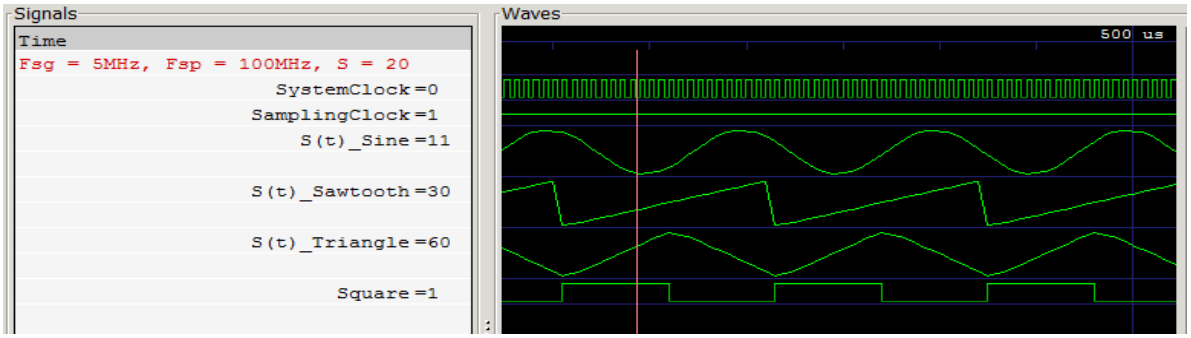


Figure 5.3(c) 5 MHz Signal generator simulation waveforms

The Figures 5.3(a), 5.3(b) and 5.3(c) show that the mathematical functions and control structures used at the functional model generated the expected wave forms.

5.3 Signal Generator Non-Functional Model

Following the previous section (5.2), the number of instructions per cycle has a bearing on the frequency of the signal, which is a performance attribute. We explore a Look-Up-Table (LUT) as a way of reducing the number of instructions per cycle. The signal values in the LUT can be read and used to generate the signal of interest, depending on the choice made by the user. This is similar to Muteithia's approach (Muteithia, 2014). However, Muteithia uses SystemC for simulation and is therefore not able to accommodate the DAC in his simulation. This is not a problem in the MSRS methodology because SystemC-AMS is used. Figure 5.4 shows the synthesized non-functional model. In addition to the LUT, we have a Phase Increment Value generator (PIV) and a Phase Accumulator (PA).

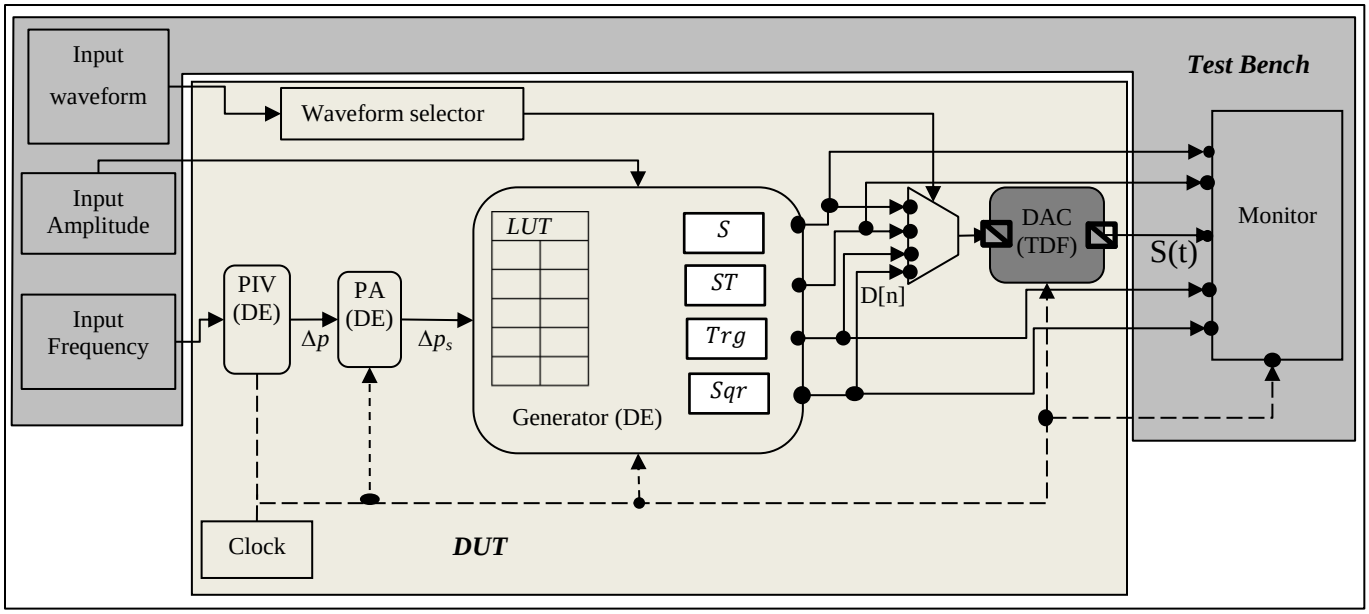


Figure 5.4: Signal Generator Non-Functional model

5.3.1 Phase Increment Value generator and Phase Accumulator

The PIV generator (whose code is in appendix B.4) generates an integral value, phase increment, Δ*P*, stored in the Phase Accumulator (PA) (whose code is in appendix B.5). At each clock cycle,

the phase increment value is added to the data previously held in the PA hence resulting to a linearly increasing digital value. The frequency of the data generated by the PA depends on three main quantities, which are; reference clock frequency, the ΔP value and j, the length of PA. The generated frequency, the reference clock frequency, the phase value and the length of the phase accumulator are related using Equation 5.5 (Vankka & Halonen, 2013).

$$f_{out} = \frac{\Delta p * f_{clk}}{2^j} \tag{5.5}$$

Where:

ΔP – is the phase increment value

j – is the number of phase accumulator bits (length of PA)

f_{clk} –is the reference clock frequency

f_{out} -is the output frequency

From the Equation 5.5, it is evident that increasing the phase increment word for a constant clock frequency and size of PA results in an increase in output frequency. The PIV generates the phase value based on the Equation 5.6 and feeds it to the PA. Equation 5.5 leads to Equation 5.6 (Vankka & Halonen, 2013).

$$\Delta p = \frac{f_{out} * 2^j}{f_{clk}} \tag{5.6}$$

The PA serves as storage of the phase value, which is connected to the Generator module. The phase value generated from the Phase Accumulator is 27 bits wide. As argued by Muteithia, the width of the phase value is the same as the width of the reference clock (100 MHz) when converted to binary value hence the choice of 27 bits (Muteithia, 2014). The PIV and the PA are discrete-event modules and therefore they are modeled using DE models.

5.3.2 Generator module

The Generator (whose code is in appendix B.6) has an array that serves as a register that stores amplitude values. The array is referred to as a LUT. For the sake of simplicity, only the sine wave is considered. The amplitude values, (LUT_m), are generated using Equation 5.7 (Vankka & Halonen, 2013).

$$LUT_m = (int)\left(A * \left(1 + sin\left(\frac{2 * \pi * M}{N}\right)\right)\right) \quad (5.7)$$

Where: LUT_m–is the LUT amplitude value generated

M – is the index of a given elements in the LUT

int – is used to convert the computed value to an integer

N– is the total number of elements in the LUT

A – is the amplitude

Table 5.3 shows some of the LUT values and their corresponding addresses used in the Generator. All the LUT values are shown as an array (given the name *amplitude[N]*) as shown on Table 5.3.The value *N* is arrived at based on the bit width of the digital signal to be generated, in this case 8-bit digital signals. Table 5.3 shows a section of LUT values for 8-bit digital signal.

Table 5.3: Some LUT values used in the signal generator

index	0	1	2	3	4	5	6	7	8	9
$LUT_{0\text{-}9}$	100	102	105	107	110	112	115	117	120	122
index	10	11	12	13	14	15	16	17	18	19
$LUT_{10\text{-}19}$	124	127	129	131	134	136	138	141	143	145
index	20	21	22	23	24	25	26	27	28	29
$LUT_{20\text{-}29}$	147	149	151	153	156	158	160	162	163	165
index	30	31	32	33	34	35	36	37	38	39
$LUT_{30\text{-}39}$	167	169	171	172	174	176	177	179	180	182

As mentioned earlier, the phase values from the PA are 27 bits wide. Once a phase value from the PA is received by the Generator module, it is shifted to the right by 19 bits. The shifting is required to truncate the 19 Least Significant Bits (LSB) and remain with only 8 Most Significant Bits (MSB). The truncation of the bits is required in order to generate phase values that range from 0 to 255 (0 to 2^8-1) which match the storage location addresses of the LUT. When the Generator receives a particular phase value, it extracts the value stored in the corresponding storage location of the LUT. The extracted values are connected to DAC (whose code is in

Appendix B.7) for conversion into analogue signals. The PIV, PA, Generator and the DAC modules are wired together in a test bench presented in Appendix B.8. The results are presented in Figures 5.5(a), 5.5(b) and 5.5(c) respectively.

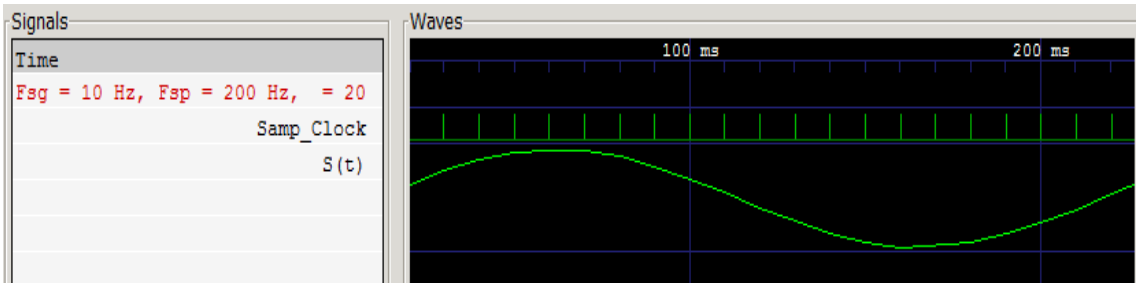


Figure 5.5(a): 10 Hz, 2 V Signal Waveforms

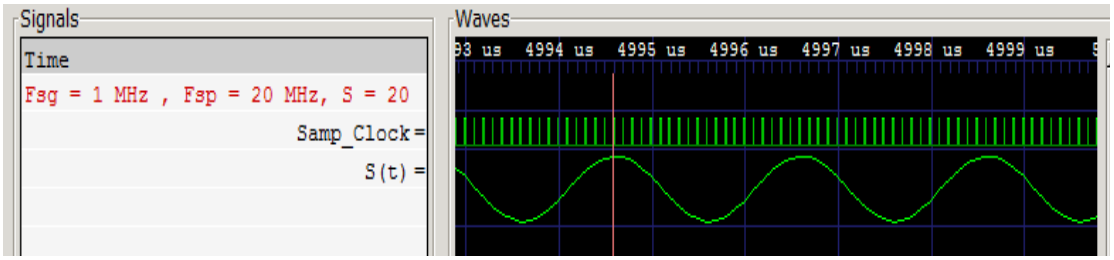


Figure 5.5(b): 1 MHz, 10 V Signal Waveforms

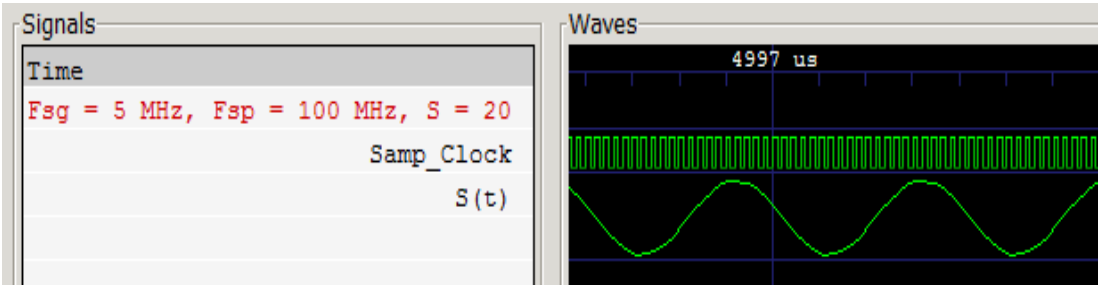


Figure 5.5(c): 5 MHz, 10 V Signal Waveforms

The Figures 5.5(a), 5.5(b) and 5.5(c) show the expected wave forms which are the same as the wave forms in Figures 5.3(a), 5.3(b) and 5.3(c). The signal wave forms simulated can be analyzed as presented in Table 5.4 for comparison purposes.

Table 5.4: Signal generator non-functional model simulation analysis

Signals freq., F_{sig}	**Values per cycle, S_c**	**Sampling freq., F_{samp}**	**Instr. to Read & Write a value, V_i**	**Instruction Freq., $F_{samp} \times V_i$**	**Remarks**
10 Hz	20	200 Hz	2	400 Hz	Minimum
1MHz	20	20 MHz	2	40 MHz	Normal
5 MHz	20	100MHz	2	200MHz	Maximum

The analyses presented in Table 5.4 shows that the instruction frequency is reduced by more than four times due to the reduced number of instructions (Read and Write), required to generate a single signal value. As explained in section 5.2 and presented in Table 5.2, to generate a single value in the functional model, nine instructions are required. This improves the upper frequency limit of the signal generator.

Muteithia (2014) separates functional and non-functional models which is similar to the approach used in this work. However, his simulation comes after the complete signal generator development is in place. This violates the preferred principle of catching errors at earliest opportunity. In the MSRS methodology, it is possible to model, simulate and evaluate the functional model before improving it with non-functional requirements and additional simulation.

5.4. Signal Generator Implementation Model

The main objective at this level is to model and simulate the DAC component that was not simulated by Muteithia (2014). The other components of the signal generator presented in the non-functional model in section 5.3, are not synthesized. This is because SystemC-AMS can simulate components at different levels of abstraction within the same system. Secondly, Muteithia demonstrated simulation using SystemC with the DAC excluded. The adapted implementation model is shown in Figure 5.6.

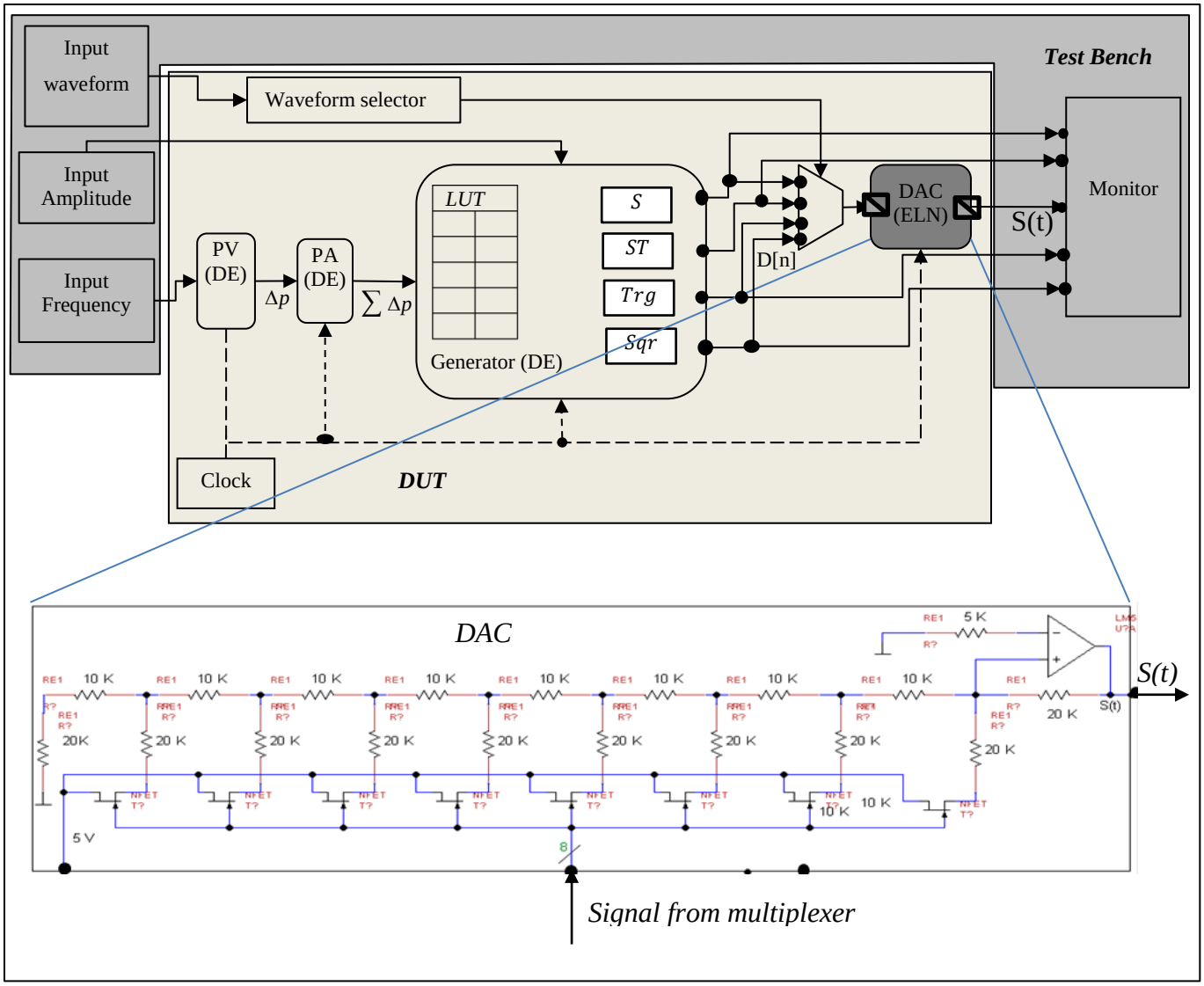


Figure 5.6: Signal Generator Implementation Model

The signal generator components presented in Figure 5.6 shows mainly a possible R-2R resistor ladder DAC in detail. The purpose of this DAC circuit is to demonstrate simulation at the analogue level of implementation. The implementation code for the DAC is presented in Appendix B.8. the implementation components of the signal generator are wired together in a test bench whose code is presented in Appendix B.9.

The signal generator system simulation has four different signals of the same frequency. The signals generated from the signal generator are sine wave, saw tooth wave, triangle wave and

square wave. The four signals generated are of 1 MHz frequency since the wave forms need to be compared with the 1 MHz signal wave forms produced by the Fusion FPGA signal generator implemented by Muteithia. The signal generator is composed of both digital and analogue components. The digital part of the generator produces the digital values of the signals which are used to sketch the digital signals, *D[n]_Sine, D[n]_Saw, D[n]_Triangle* presented in Figure 5.7. The analog part of the signal generator is composed of the DAC which converts the digital signals to analog signals. The signals generated from the DAC, *S(t)_Sine, S(t)_Saw, S(t)_Triangle* are presented in Figure 5.7.

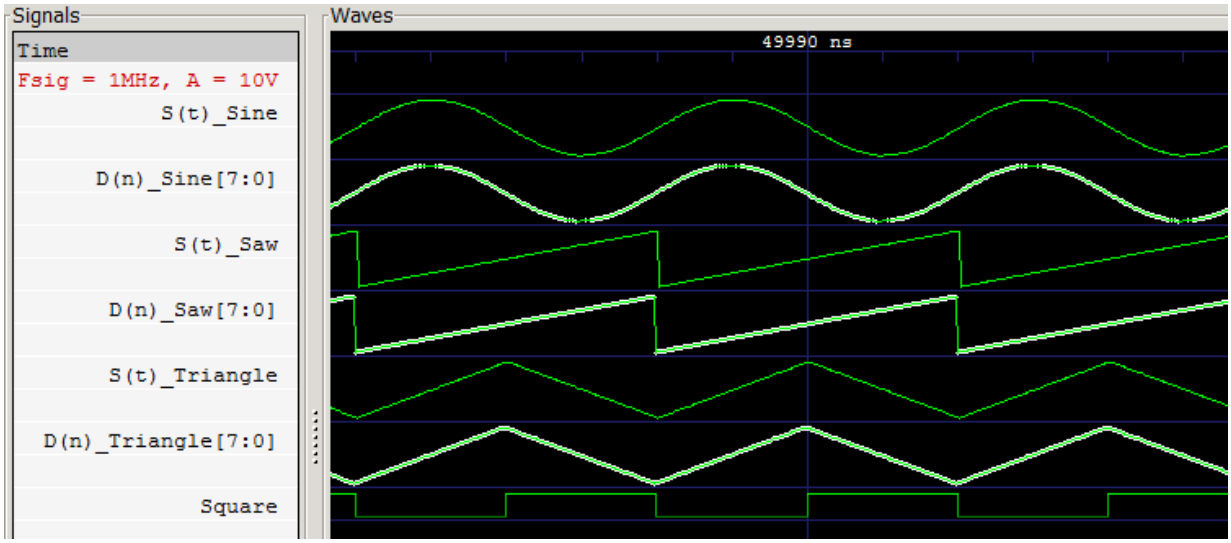


Figure 5.7: Signal Generator Implementation Model Waveforms

Figure 5.7 presents both digital signals and their analogue equivalent signals. The purpose of presenting the digital signals and analog signals waveforms is to compare them in order to justify the operation of the SystemC-AMS modeled and simulated DAC. For each wave form, comparing the digital and analogue signals, it is evident that they are the same. This shows that that the DAC component modeled and simulated using SystemC-AMS is viable.

The simulated signal generator results are compared with waveforms generated by a signal generator system developed and implemented in a Fusion FPGA by Muteithia. Therefore, Muteithia's test results are presented in the next section (5.5).

5.5 FPGA Signal Generator System Waveforms

The waveforms from the FPGA signal generator presented here are for comparison purpose. In his work, Muteithia simulated only the digital components of his signal generator using SystemC. He could not simulate the DAC to be used in his development using SystemC because it can only model and simulate discrete event signals (Black & Donovan, 2004), (Bhasker, 2002), hence he ended up implementing the DAC without simulation (Muteithia, 2014). The FPGA signal generator waveforms from the DAC output in his implementation work are presented in Figures 5.8(a), 5.9(a), 5.10(a) and 5.11(a). For the purpose of comparison, the simulated waveforms for the respective signals are presented in Figures 5.8(b), 5.9(b), 5.10(b) and 5.11(b).

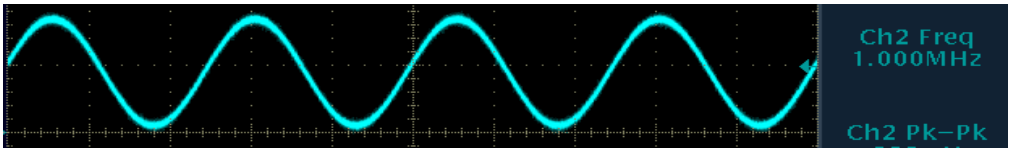


Figure 5.8(a): 1 MHz FPGA Sine wave (Muteithia, 2014)

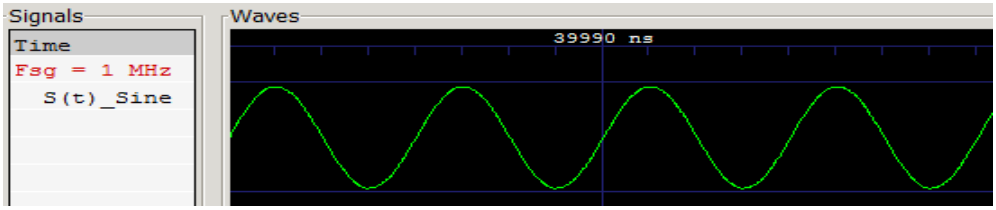


Figure 5.8(b): 1 MHz Simulated Sine Wave

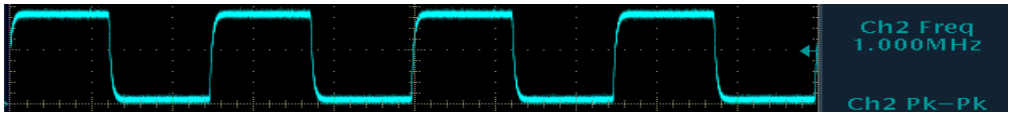


Figure 5.9(a): 1 MHz FPGA Square wave (Muteithia, 2014)

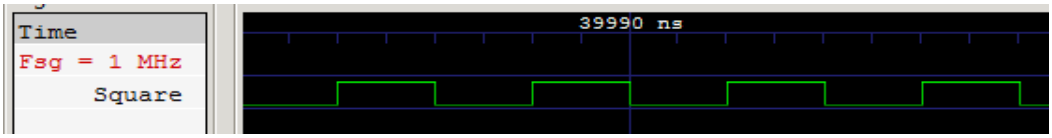


Figure 5.9(b): 1 MHz Simulated Square wave

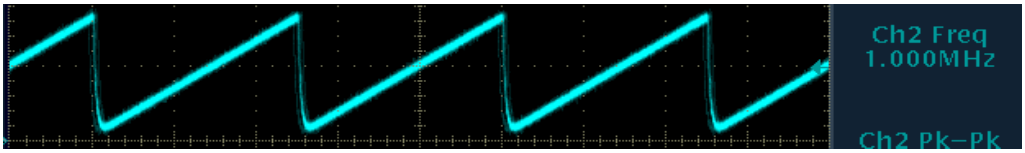


Figure 5.10(a):1 MHz FPGA Saw tooth wave (Muteithia, 2014)

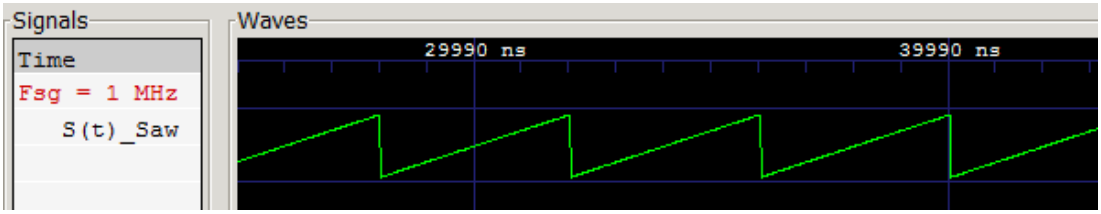


Figure 5.10(b):1 MHz Simulated Saw tooth wave

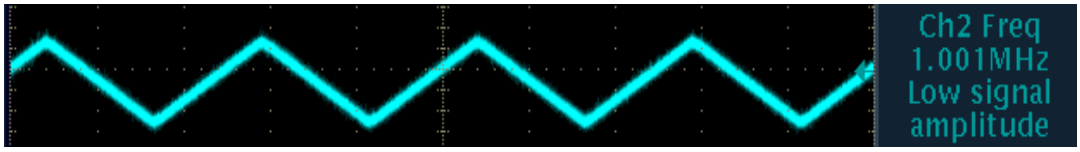


Figure 5.11(a):1 MHz FPGA Triangular wave (Muteithia, 2014)

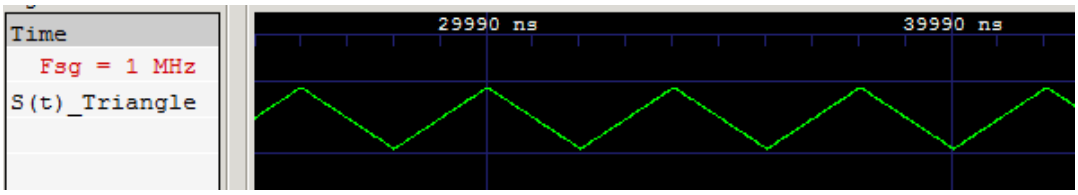


Figure 5.11(b):1 MHz Simulated Triangular wave

As can be seen, the test results of the FPGA signal generator are replicated by the simulated signal generator. Therefore, these results are used to support the claim that the MSRS methodology is effective.

CHAPTER SIX

CONCLUSION AND RECOMMENDATIONS FOR FUTURE WORK

6.1 Conclusion

A Model-Simulate-Refine-Synthesis (MSRS) methodology for modeling and simulation from functional level down to the implementation level has been developed and presented. The methodology helps the developer to model and simulate both analogue and digital components at functional, non-functional and implementation levels of abstraction by generating functional, non-function and implementation models corresponding to each level respectively, which is valuable for developing E-AMS systems.

The methodology has been verified through modeling and simulation of an oscilloscope. The simulation results of the functional model are used to validate the model while for the implementation model are used to verify the model against the functional model. Comparing the waveforms from the functional model and the waveforms from the implementation model, it is found that they are similar. Therefore, this makes the methodology developed viable in modeling and simulation of E-AMS systems.

Further, the methodology has been validated through modeling and simulation of a signal generator. The modeled and simulated signal generator simulation results are compared with test results obtained from a real similar signal generator developed and implemented in a Fusion FPGA. Comparing the waveforms, it is evident that the waveforms from the simulated signal generator and the waveforms from the FPGA signal generator are similar. Therefore, the methodology has been validated.

Therefore, the methodology has been verified and validated and gives satisfactory results, the objectives of the work have been achieved.

The MSRS methodology developed has several strengths:

- The methodology provides a systematic way to develop E-AMS systems from the functional level down to the implementation level.

- The methodology allows multiple development paradigms such as evolutionary, iterative, and incremental (Larman & Basili, 2003).
- It synchronizes system development and learning. Innovative system development usually intertwines learning and system development phases.
- By simulation at every level of abstraction, there is a good opportunity to catch many errors at the earliest opportunity.

On the other hand, the MSRS methodology has some drawbacks:

- Knowledge of SystemC-AMS is required. This is a heavy undertaking considering that SystemC-AMS is built on top of C++ and extends SystemC.
- At the time of this work, there are no specific component libraries from manufacturers. This hinders generation of blueprint models for implementation.

6.2 Recommendations for future work

Based on the drawbacks of the methodology, then:

- Future work could explore combining modeling at the system level with automation to automatically generate blueprints (Mosterman & Vangheluwe, 2004). This would avoid knowledge of C++, SystemC and SystemC-AMS.
- A component library standard to ensure interoperability among component libraries from different manufacturers. This will facilitate development of a blueprint as the last stage of the implementation model.
- Some research may be done to use the MSRS methodology in designing and fabricating a real E-AMS system.

REFERENCES

Ashenden, P. J. (2010). *The designer's guide to VHDL* (Vol. 3). Morgan Kaufmann.

Barnasconi, M., Einwich, K., Grimm, C., Maehne, T., & Vachoux, A. (2011). Advancing the SystemC Analog/Mixed-Signal (AMS) Extensions. *Open SystemC Initiative (OSCI)*.

Barnasconi, M., Grimm, C., Damn, M., Enwich, K., Louërat, M. M., Mähne, T., & Vachoux, A. (2010). SystemC AMS extensions user's guide. *Open SystemC Initiative (OSCI). Mar*, *8*, 14-72.

Bhasker, J. (2002). A systems "Primer". http://read.pudn.com/downloads92/ebook/359261/A SystemC Primer.pdf , Accessed: 23 Feb 2015

Black, D. C., Donovan, J., Bunton, B., & Keist, A. (2009). *SystemC: From the Ground Up: From the Ground Up* (Vol. 71). Springer Science & Business Media.

Broeders, J. Z. M. (2010). *Extracting behavior and dynamically generated hierarchy from SystemC models* (Doctoral dissertation, TU Delft, Delft University of Technology).

Cai, L., & Gajski, D. (2003). Transaction level modeling in system level design.*Center for Embedded Computer Systems*.

Calva, C. A., Rocha, M. F., Orozco, L., Gaso, M. R., Osnaya, M. R., Navarrete, R., & Solis, C. (2012). Design of a Low Cost Electronic System for Automotive Steering Controlling. *International Journal of Computer and Communication Engineering*, *1*(4), 313.

Donlin, A. (2004, September). Transaction level modeling: flows and use models. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis* (pp. 75-80). ACM.

Dowell, R. I. (2011). Hijacked by SPICE. *IEEE Solid-State Circuits Magazine*,*2*(3), 13-15.

Gajski, D. D., & Kuhn, R. H. (1983). Guest editors' introduction: New VLSI tools. *Computer*, *16*(12), 11-14.

Gajski, D. D., Abdi, S., Gerstlauer, A., & Schirner, G. (2009). *Embedded system design: modeling, synthesis and verification*. Springer Science & Business Media.

Ghenassia, F. (2005). *Transaction-level modeling with SystemC* (pp. 153-183). Dordrecht, The Netherlands: Springer.

Gray, N. (2006). ABCs of ADCs Analog-to-Digital Converter Basics. *Gray, N., Data Conversion System, Staff Applications Engineer, National Semiconductor Corp*, 1-64.

Grimm, C., Barnasconi, M., Vachoux, A., & Einwich, K. (2008, June). An introduction to modeling embedded analog/mixed-signal systems using SystemC AMS extensions. In *DAC2008 International Conference*.

Jensen, R. W. (1966). Charge Control Transistor Model for the IBM Electronic Circuit Analysis Program. *Circuit Theory, IEEE Transactions on*, *13*(4), 428-437.

Jeruchim, M. C., Balaban, P., & Shanmugan, K. S. (2006). *Simulation of communication systems: modeling, methodology and techniques*. Springer Science & Business Media.

Karnane, K., Curtis, G., & Goering, R. (2009). Solutions for mixed-signal SoC verification. *Cadence Design Systems*.

Kelemenová, T., Kelemen, M., Miková, Ľ., Maxim, V., Prada, E., Lipták, T., & Menda, F. (2013). Model based design and HIL simulations. *American Journal of Mechanical Engineering*, *1*(7), 276-281.

Kodi, A. K., & Louri, A. (2008). Optisim: A system simulation methodology for optically interconnected HPC systems. *IEEE micro*, (5), 22-36.

Larman, C., & Basili, V. R. (2003). Iterative and incremental development: A brief history. *Computer*, (6), 47-56.

Larman, C., & Basili, V. R. (2003). Iterative and incremental development: A brief history. *Computer*, (6), 47-56.

Lyons, R. G. (Ed.). (2012). *Streamlining digital signal processing: a tricks of the trade guidebook*. John Wiley & Sons.

Mähne, T. (2011). Efficient Modelling and Simulation Methodology for the Design of Heterogeneous Mixed-Signal Systems on Chip.

Mischkalla, F., He, D., & Mueller, W. (2010, March). Closing the gap between UML-based modeling, simulation and synthesis of combined HW/SW systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*(pp. 1201-1206). IEEE.

Montealegre Lobo, L., Dufour, C., & Mahseredjian, J. (2013, September). Real-time simulation of More-Electric Aircraft power systems. In *Power Electronics and Applications (EPE), 2013 15th European Conference on* (pp. 1-10). IEEE.

Mosterman, P. J., & Vangheluwe, H. (2004). Computer automated multi-paradigm modeling: An introduction. *Simulation*, *80*(9), 433-450.

Muteithia, W. M. (2014). *Design And Development Of An FPGA Based DDFS Signal Generator* (MSc. Thesis, University of Nairobi).

Najy, R. J. (2013). The Role of ComputerAided Design (CAD) in the Manufacturing and Digital Control (CAM). *Contemporary Engineering Sciences*,*6*, 297-312.

Pêcheux, F., Allard, B., Lallement, C., Vachoux, A., & Morel, H. (2005). Modeling and simulation of multi-discipline systems using bond graphs and VHDL-AMS.

In *Proceedings of the International Conference on Bond Graph Modeling and Simulation (ICBGM)* (No. LSM-CONF-2005-003).

Pedroni, V. A. (2004). *Circuit design with VHDL*. MIT press.

Ptolemaeus, C. (2014). *System Design, Modeling, and Simulation: Using Ptolemy II*. Berkeley, CA, USA: Ptolemy. org.

Roberts Jr, B. D., & Harbourt, C. O. (1967). Computer models of the field-effect transistor. *Proceedings of the IEEE*, *55*(11), 1921-1929.

Saxena, R. S., Panwar, A., Semwal, S. K., Rana, P. S., Gupta, S., & Bhan, R. K. (2012). PSPICE circuit simulation of microbolometer infrared detectors with noise sources. *Infrared Physics & Technology*, *55*(6), 527-532.

Sinha, R., Paredis, C. J., Liang, V. C., & Khosla, P. K. (2001). Modeling and simulation methods for design of engineering systems. *Journal of Computing and Information Science in Engineering*, *1*(1), 84-91.

Szermer, M., Daniel, M., & Napieralski, A. (2003, February). Modeling and simulation sigma-delta analog to digital converters using VHDL-AMS. In *CAD Systems in Microelectronics, 2003. CADSM 2003. Proceedings of the 7th International Conference. The Experience of Designing and Application of* (pp. 331-333). IEEE.

Thomas, D. E., & Moorby, P. R. (2002). *The Verilog® Hardware Description Language* (Vol. 2). Springer Science & Business Media.

Topper, J. S., & Horner, N. C. (2013). Model-Based Systems Engineering in Support of Complex Systems Development. *Johns Hopkins Apl Technical Digest*, *32*(1).

Vachoux, A. (1998). Analog and mixed-signal extensions to VHDL. In *Analog VHDL* (pp. 97-112). Springer US.

Vachoux, A., Grimm, C., & Einwich, K. (2004, January). Towards analog and mixed-signal SOC design with systemC-AMS. In *Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on* (pp. 97-102). IEEE.

Vankka, J., & Halonen, K. A. (2013). *Direct digital synthesizers: theory, design and applications* (Vol. 614). Springer Science & Business Media.

Vasilevski, M., Pecheux, F., Aboushady, H., & De Lamarre, L. (2007, September). Modeling heterogeneous systems using SystemC-AMS case study: A wireless sensor network node. In *Behavioral Modeling and Simulation Workshop, 2007. BMAS 2007. IEEE International* (pp. 11-16). IEEE.

Wilson, P., & Mantooth, H. A. (2013). *Model-based Engineering for Complex Electronic Systems: Techniques, Methods and Applications*. Newnes.

Zorzi, M., Franze, F., & Speciale, N. (2003). Construction of VHDL-AMS simulator in MatlabTM. In *Proceedings of the 2003 International Workshop on Behavioral Modeling* (pp. 113-117).

APPENDIX A: OSCILLOSCOPE CODES

A.1: Modulator Functional Model Code

```
#include "systemc.h"
#include "systemc-ams.h"

SC_MODULE(modulator)
{
	//TDF input ports
	sca_tdf::sca_in<double>		sig_in;
	sca_tdf::sca_in<double>		k_in;

	//TDF output port
	sca_tdf::sca_out<double>	cond_sign_out;

	//TDF to LSF converter ports
	sca_lsf::sca_tdf_source		conv_sign;
	sca_lsf::sca_tdf_sink		conv_cond_sign;

	//gain
	sca_lsf::sca_gain		gain1;

	//linking lsf signals
	sca_lsf::sca_signal		sign_link;
	sca_lsf::sca_signal		cond_link;

	//Module constructor
	modulator(sc_core::sc_module_name, double k1 = 100.0)
		: sig_in("sig_in"), k_in("k_in"), gain1("gain1", k1),
		cond_sign_out("cond_sign_out"),
		conv_sign("conv_sign"), conv_cond_sign("conv_cond_sign")
	{
		//TDF to LSF conversion
		conv_sign.inp(sig_in);
		conv_sign.y(sign_link);

		//gain connection
		gain1.x(sign_link);
		gain1.y(cond_link);

		//LSF to TDF conversion
		conv_cond_sign.x(cond_link);
		conv_cond_sign.outp(cond_sign_out);
	}
};
```

A.2: Sampler Functional Model Code

```
#include "systemc.h"
#include "systemc-ams.h"

SCA_TDF_MODULE(sampler)
{
```

```
	//TDF ports
	sca_tdf::sca_in<double>		ksin;
	sca_tdf::sca_in<bool>		plsin;
	sca_tdf::sca_out<double>	smpld;

	//module constructor
	sampler(sc_core::sc_module_name)
		: ksin("ksin"), plsin("plsin"), smpld("smpld")
	{}
	void processing()
	{
		//Sampling when pulse is high else write 0 to the port
		if (plsin.read() == 1)
		{
			smpld.write(ksin.read());
		}
		else
		{
			smpld.write(0);
		}
	}
};
```

A.3: Hold Functional Model Code

```
#include "systemc.h"
#include "systemc-ams.h"

SCA_TDF_MODULE(thefilter)
{
	//TDF ports
	sca_tdf::sca_in<double>		sigin;
	sca_tdf::sca_in<bool>		plsin;
	sca_tdf::sca_out<double>	sigout;

	//module constructor
	thefilter(sc_core::sc_module_name)
	: sigin("sigin"), plsin("plsin"), sigout("sigout")
	{}

	void processing()
	{
		//holds the sampled value when the pulse is high
		if (plsin.read() == 1)
		{
			//Holds the value until the next sample
			sigout.write(sigin.read());

		}
	}
};
```

A.4: Functional ADC Model Code

```
#include "systemc.h"
#include "systemc-ams.h"

SCA_TDF_MODULE(theadc)
{
      //TDF input ports
      sca_tdf::sca_in<double> anlgsigin;
      sca_tdf::sca_in<bool>   pulsin;

      //TDF-DE converter output port
      sca_tdf::sca_de::sca_out<sc_uint<8>>      adcout;

      //variable declaration
      sc_uint<8>  adcval;
      float       samp;
      int         intsampval, i, r, dev;

      //module constructor
      theadc(sc_core::sc_module_name)
            : anlgsigin("anlgsigin"), adcout("adcout"),pulsin("pulsin")
      {}
      void processing()
      {
            //Process done when the pulse is high
            if (pulsin.read() == 1)
            {
                  //Sampling
                  samp = (anlgsigin.read());
                  intsampval = (int) (samp);

                  //conversion to binary
                  for(i=0; i<8; i++)
                  {
                        r = intsampval % 2;
                        dev = intsampval / 2;
                        adcval[i] = r;
                        intsampval = dev;
                  }

                  //Output the digital signal
                  adcout.write(adcval);
            }
      }
};
```

A.5: Oscilloscope Functional model Test Bench code

```
//include oscilloscope modules
#include "driver.h"
#include "signalsource.cpp"
#include "enableunit.cpp"
#include "controlunit.cpp"
#include "modulator.cpp"
```

```
#include "sampler.cpp"
#include "adc.cpp"
#include "hold.cpp"

int sc_main(int argc, char *argv[])
{
      sc_report_handler::set_actions("/IEEE_Std_1666/deprecated",
SC_DO_NOTHING);

      //declaraing signals to link different modules
      //1. DE signals
      sc_signal<double>             t_frequency;
      sc_signal<double>             t_ampltd;
      sc_signal<double>             t_enbl_ctrl;
      sc_signal<sc_uint<8> >        adcsign_8;

      //2. TDF signals
      sca_tdf::sca_signal<double>   t_enbl_s;
      sca_tdf::sca_signal<double>   t_enbl_k;
      sca_tdf::sca_signal<double>   t_enbl_pls;
      sca_tdf::sca_signal<bool>     t_pls;
      sca_tdf::sca_signal<double>   t_sine_out;
      sca_tdf::sca_signal<double>   t_det_k;
      sca_tdf::sca_signal<double>   t_k_val;
      sca_tdf::sca_signal<double>   t_cond_sign;
      sca_tdf::sca_signal<double>   t_sampld_sign;
      sca_tdf::sca_signal<double>   t_filtered_sign;

      // declare a time constant for the system clock
      const sc_time t_period(20, SC_NS);
      sc_clock clk("clk", t_period);

      //Add modules
      //1. Driver module
      syst_driver dr("Driver_Module");
      dr.d_frequency(t_frequency);
      dr.d_ampl(t_ampltd);
      dr.clock(clk);
      dr.enbl_out(t_enbl_ctrl);

      //2. Enable unit module
      enbl en("enable_Unit");
      en.infrq(t_frequency);
      en.enabl(t_enbl_ctrl);
      en.enb_s(t_enbl_s);
      en.enb_k(t_enbl_k);
      en.enb_pls_gen(t_enbl_pls);

      //3. Signal source module
      sinewv sr("Signal_source");
      sr.inpenb(t_enbl_s);
      sr.inpfrq(t_frequency);
      sr.out(t_sine_out);
      sr.det_k(t_det_k);
```

```
	//4. Modulator module
	theconditioner con("Modulator");
	con.sig_in(t_sine_out);
	con.k_in(t_k_val);
	con.cond_sign_out(t_cond_sign);

	//5. Sampler module
	sampl sp("Sampler");
	sp.plsin(t_pls);
	sp.ksin(t_cond_sign);
	sp.smpld(t_sampld_sign);

	//6. Hold module
	thefilter fl("HoldModule");
	fl.plsin(t_pls);
	fl.sigin(t_sampld_sign);
	fl.sigout(t_filtered_sign);

	//7. ADC module
	theadc d("ADC");
	d.pulsin(t_pls);
	d.anlgsigin(t_filtered_sign);
	d.adcout(adcsign_8);

	// Tracing waveforms
	//1. creating a file to which values are to be stored
	sca_util::sca_trace_file *anatf =
	sca_util::sca_create_vcd_trace_file("WaveForms");

	//2. Storing different values in the file
	cout << "Start tracing waveforms " << endl;
	sca_util::sca_trace(anatf, t_sine_out, "S(t)");
	sca_util::sca_trace(anatf, t_cond_sign, "KS(t)");
	sca_util::sca_trace(anatf, t_pls, "Pulses");
	sca_util::sca_trace(anatf, t_sampld_sign, "KS(nT)");
	sca_util::sca_trace(anatf, t_filtered_sign, "KH(t)S(nT)");
	sca_util::sca_trace(anatf, adcsign_8, "D8(n)");

	//simulation duration to be taken
	sc_start(100, SC_US);

	//closing the file
	sca_util::sca_close_vcd_trace_file(anatf);
	cout << "Finished tracing waveforms" << endl;
	return (0);
}
```

A.6: Control Unit Module Code

```
#include "systemc.h"
#include "systemc-ams.h"
SCA_TDF_MODULE(thectrl)
{
	//TDF ports
```

```
sca_tdf::sca_in<double>       enabl;
sca_tdf::sc_in<double>        inpfrq;
sca_tdf::sc_in<double>        inpamp;
sca_tdf::sca_out<double>      k_valout;
sca_tdf::sca_out<bool>        pulsout;
//LUT
float lut[6] = {0.01,0.1,1,10,100,1000,0};
float ampltd;
//Module constructor
thectrl(sc_core::sc_module_name,
      sca_core::sca_time tm_ = sca_core::sca_time(10.0,
      sc_core::SC_PS))
    : inpfrq("infrq"), enabl("enabl"), k_valout("k_valout"), tm(tm_),
       pulsout("pulsout"), inpamp("inpamp")
      {}
//setting time steps
void set_attributes()
{
      set_timestep(tm);
}
void processing()
{
      //process once enabled
      if (enabl.read() == 1)
      {
      if(inpfrq.read()==0)
      {
            pulsout.write(1);
      }
      else
      {
            //Generating the sampling clock based on the system clock
            //and signal frequency
            x = x + 1;
            fsyst = 100000000000; //system clk frequency
            stot = (int)(fsyst / inpfrq.read());
            s = 20;     //samples required in a single cycle
            n = (int)(stot / s);
            r = x % n;
            if (r == 0)
            {
                  pulsout.write(1); //generate a HIGH state
            }
            else
            {
                  pulsout.write(0); //generate a LOW state
            }
            //Choosing k value depending on the signal amplitude
            ampltd = inpamp.read();
            if(ampltd >= 33)
            {
                  k_valout.write(lut[0]);
            }
            else if(ampltd >= 3.3)
            {
```

```
                        k_valout.write(lut[1]);
                  }
                  else if(ampltd >= 0.33)
                  {
                        k_valout.write(lut[2]);
                  }
                  else if(ampltd >= 0.033)
                  {
                        k_valout.write(lut[3]);
                  }
                  else if(ampltd >= 0.0033)
                  {
                        k_valout.write(lut[4]);
                  }
                  else if(ampltd >= 0.00033)
                  {
                        k_valout.write(lut[5]);
                  }
                  else
                  {
                        k_valout.write(lut[6]);
                  }
                  }
            }
      }
private:
      double stot, s, fsyst;
      int x, r, n;
      double freqval, sigperiod, sampperiod;
      sca_core::sca_time tm;
};
```

A.7: ADC Non Functional Module Code

```
#include "systemc.h"
#include "systemc-ams.h"
#define A 3
SCA_TDF_MODULE(theadc)
{
      //TDF input ports
      sca_tdf::sca_in<double> anlgsigin;
      sca_tdf::sca_in<bool>   pulsin;
      //TDF-DE output ports
      sca_tdf::sca_de::sca_out<sc_uint<10>>     adcout10;
      sca_tdf::sca_de::sca_out<sc_uint<8>>      adcout8;
      sca_tdf::sca_de::sca_out<sc_uint<6>>      adcout6;
      sca_tdf::sca_de::sca_out<sc_uint<4>>      adcout4;
      sca_tdf::sca_de::sca_out<sc_uint<2>>      adcout2;
      //Variable declaration
      sc_uint<10> adcval10;
      sc_uint<8>  adcval8;
      sc_uint<6>  adcval6;
      sc_uint<4>  adcval4;
      sc_uint<2>  adcval2;
```

```
int   samp;
int   intsampval10, i10, r10, dev10;
int   intsampval8, i8, r8, dev8;
int   intsampval6, i6, r6, dev6;
int   intsampval4, i4, r4, dev4;
int   intsampval2, i2, r2, dev2;
//Module constructor
theadc(sc_core::sc_module_name)
      : anlgsigin("anlgsigin"),
      adcout10("adcout10"),
      adcout8("adcout8"),
      adcout6("adcout6"),
      adcout4("adcout4"),
      adcout2("adcout2"),
      pulsin("pulsin")
{}
void processing()
{
      //process when the pulse is in H state
      if (pulsin.read() == 1)
      {
            //sampling and conversion to digital signals
            samp = (int)(anlgsigin.read());
            //Determine equivalent integral value depending on the ADC
            //bits required
            intsampval10 = (int) (samp * 1024 / (2 * A)); //for 10 bits
            intsampval8 = (int) (samp * 256 / (2 * A)); //for 8 bits
            intsampval6 = (int) (samp * 64 / (2 * A)); //for 6 bits
            intsampval4 = (int) (samp * 16 / (2 * A)); //for 4 bits
            intsampval2 = (int) (samp * 4 / (2 * A)); //for 2 bits
            //10 bit conversion
            for(i10=0; i10<10; i10++)
            {
                  r10 = intsampval10 % 2;
                  dev10 = intsampval10 / 2;
                  adcval10[i10] = r10;
                  intsampval10 = dev10;
            }
            //8 bit conversion
            for(i8=0; i8<8; i8++)
            {
                  r8 = intsampval8 % 2;
                  dev8 = intsampval8 / 2;
                  adcval8[i8] = r8;
                  intsampval8 = dev8;
            }
            //6 bit conversion
            for(i6=0; i6<6; i6++)
            {
                  r6 = intsampval6 % 2;
                  dev6 = intsampval6 / 2;
                  adcval6[i6] = r6;
                  intsampval6 = dev6;
            }
```

```
                  //4 bit conversion
                  for(i4=0; i4<4; i4++)
                  {
                        r4 = intsampval4 % 2;
                        dev4 = intsampval4 / 2;
                        adcval4[i4] = r4;
                        intsampval4 = dev4;
                  }
                  //2 bit conversion
                  for(i2=0; i2<2; i2++)
                  {
                        r2 = intsampval2 % 2;
                        dev2 = intsampval2 / 2;
                        adcval2[i2] = r2;
                        intsampval2 = dev2;
                  }
                  //output the digital signals
                  adcout10.write(adcval10);
                  adcout8.write(adcval8);
                  adcout6.write(adcval6);
                  adcout4.write(adcval4);
                  adcout2.write(adcval2);
            }
      }
};
```

A.8: None-Functional Model Test Bench Code

```
//include modules
#include "driver.h"
#include "signalsource.cpp"
#include "enableunit.cpp"
#include "controlunit.cpp"
#include "modulator.cpp"
#include "sampler.cpp"
#include "adc.cpp"
#include "hold.cpp"

int sc_main(int argc, char *argv[])
{
      sc_report_handler::set_actions("/IEEE_Std_1666/deprecated",
SC_DO_NOTHING);
      //defining signals to connect modules
      sc_signal<double>       t_frequency;
      sc_signal<double>       t_ampltd;
      sc_signal<double>       t_enbl_ctrl;
      sca_tdf::sca_signal<double> t_enbl_s;
      sca_tdf::sca_signal<double> t_enbl_k;
      sca_tdf::sca_signal<double> t_enbl_pls;
      sca_tdf::sca_signal<bool>     t_pls;
      sca_tdf::sca_signal<double>   t_sine_out;
      sca_tdf::sca_signal<double>   t_det_k;
      sca_tdf::sca_signal<double>   t_k_val;
      sca_tdf::sca_signal<double>   t_cond_sign;
```

```
sca_tdf::sca_signal<double>   t_sampld_sign;
sca_tdf::sca_signal<double>   t_filtered_sign;
sc_signal<sc_uint<10> >       adcsign_10;
sc_signal<sc_uint<8> >        adcsign_8;
sc_signal<sc_uint<6> >        adcsign_6;
sc_signal<sc_uint<4> >        adcsign_4;
sc_signal<sc_uint<2> >        adcsign_2;

// declare a time constant for the system clock
const sc_time t_period(100, SC_PS);
sc_clock clk("clk", t_period);

//Driver module
syst_driver dr("Driver_Module");
dr.d_frequency(t_frequency);
dr.d_ampl(t_ampltd);
dr.clock(clk);
dr.enbl_out(t_enbl_ctrl);

//Eneble unit module
enbl en("enable_Unit");
en.infrq(t_frequency);
en.enabl(t_enbl_ctrl);
en.enb_s(t_enbl_s);

//Signal source module
sinewv sr("Signal_source");
sr.inpenb(t_enbl_s);
sr.inpfrq(t_frequency);
sr.inpampltd(t_ampltd);
sr.out(t_sine_out);

//Control unit module
thectrl ct("TheControlUnit");
ct.enabl(t_enbl_s);
ct.inpfrq(t_frequency);
ct.inpamp(t_ampltd);
ct.k_valout(t_k_val);
ct.pulsout(t_pls);

//modulator module
themodulator con("modulator");
con.sig_in(t_sine_out);
con.k_in(t_k_val);
con.cond_sign_out(t_cond_sign);

//sampler module
sampl sp("Sampler");
sp.plsin(t_pls);
sp.ksin(t_cond_sign);
sp.smpld(t_sampld_sign);

//Hold module
thehold fl("Hold");
fl.plsin(t_pls);
```

```
      fl.sigin(t_sampld_sign);
      fl.sigout(t_filtered_sign);

      //ADC module
      theadc d("ADC");
      d.pulsin(t_pls);
      d.anlgsigin(t_filtered_sign);
      d.adcout10(adcsign_10);
      d.adcout8(adcsign_8);
      d.adcout6(adcsign_6);
      d.adcout4(adcsign_4);
      d.adcout2(adcsign_2);

      // Tracing waveforms
      sca_util::sca_trace_file *anatf =
sca_util::sca_create_vcd_trace_file("WaveForms");

      cout << "Start tracing waveforms " << endl;
      sca_util::sca_trace(anatf, t_sine_out, "S(t)");
      sca_util::sca_trace(anatf, t_k_val, "K");
      sca_util::sca_trace(anatf, t_cond_sign, "KS(t)");
      sca_util::sca_trace(anatf, t_sampld_sign, "KS(nT)");
      sca_util::sca_trace(anatf, t_filtered_sign, "KH(t)S(nT)");
      sca_util::sca_trace(anatf, adcsign_10, "D10(n)");
      sca_util::sca_trace(anatf, adcsign_8, "D8(n)");
      sca_util::sca_trace(anatf, adcsign_6, "D6(n)");
      sca_util::sca_trace(anatf, adcsign_4, "D4(n)");
      sca_util::sca_trace(anatf, adcsign_2, "D2(n)");

      //Simulation period
      sc_start(60, SC_US);

      sca_util::sca_close_vcd_trace_file(anatf);
      cout << "Finished tracing waveforms" << endl;

      return (0);
}
```

A.9: Buffer Implementation Model Code

```
#include "systemc.h"
#include "systemc-ams.h"

SC_MODULE(buff)
{
      //Defining ports
      sca_tdf::sca_in<double>       sigin;
      sca_tdf::sca_out<double>      sigout;
      sca_tdf::sca_in<bool>         swtch10;
      sca_tdf::sca_in<bool>         swtch100;
      sca_eln::sca_tdf_vsource      sigtdf2eln; // TDF to ELN converter
      sca_eln::sca_tdf_vsink        eln2tdf;    // ELN to TDF converter
      sca_eln::sca_nullor           opamp1;     //Operational amplifier
      sca_eln::sca_r                r1, r2, r3; //Resistors
```

```
      //ELN-TDF signal controlled switches
      sca_eln::sca_tdf::sca_rswitch rswth10, rswth100;
      //Module constructor
      buff(sc_core::sc_module_name)
            : sigin("sigin"), sigout("sigout"),
            sigtdf2eln("sigtdf2eln"), eln2tdf("eln2tdf"),
            opamp1("opamp1"), r1("r1", 3.3e6), r2("r2", 336e3),
             r3("r3", 33.3e2),
            swtch10("swtch10"), swtch100("swtch100"),
            rswth10("rswth10", 0.0, 1e20, 0),
            rswth100("rswth100", 0.0, 1e20, 0),
            gnd("gnd")
      {
            //convert the TDF to ELN signal
            sigtdf2eln.inp(sigin);
            sigtdf2eln.p(n1);
            sigtdf2eln.n(gnd);

            //ELN signals
            r1.n(n1);   r1.p(n2);

            //Attenuate by 10
            rswth10.p(n2);
            rswth10.ctrl(swtch10);
            rswth10.n(n3);
            r2.n(n3);   r2.p(gnd);

            //Attenuate by 100
            rswth100.p(n2);
            rswth100.ctrl(swtch100);
            rswth100.n(n4);
            r3.n(n4);   r3.p(gnd);

            //operational amplifier
            opamp1.nip(n2);
            opamp1.nin(n5);
            opamp1.nop(n5);
            opamp1.non(gnd);

            eln2tdf.p(n5);
            eln2tdf.n(gnd);
            eln2tdf.outp(sigout); //output connection
      }
public:
      sca_eln::sca_node n1, n2, n3, n4, n5;
      sca_eln::sca_node_ref   gnd;
};
```

A.10: Modulator Implementation Model Code

```
#include "systemc-ams.h"
#include "systemc.h"
```

```
SC_MODULE(modgain)
{
      //Module Ports
      sca_tdf::sca_in<double> vin;
      sca_tdf::sca_in<bool>   amp1;
      sca_tdf::sca_in<bool>   amp10;
      sca_tdf::sca_in<bool>   amp100;
      sca_tdf::sca_in<bool>   amp1000;
      sca_eln::sca_terminal   vout;

      //Resistors
      sca_eln::sca_r          r, rc, rf;

      //ELN-TDF controlled variable resistor
      sca_eln::sca_r                r1, r10, r100, r1000;

      //ELN-TDF controlled switches
      sca_eln::sca_tdf::sca_rswitch rswt1, rswt10, rswt100, rswt1000;

      //operational amplifiers
      sca_eln::sca_nullor     opamp1, opamp2;

      //TDF to ELN converter
      sca_eln::sca_tdf::sca_vsource convs;

      //Module constructor
      SC_CTOR(modgain)
            : vin("vin"), vout("vout"), opamp1("opamp1"), gnd("gnd"),
            rc("rc", 1e3), rf("rf", 1e3), opamp2("opamp2"), convs("conv"),
            r("r", 1e3), r1("r1", 1e3), r10("r10", 10e3), r100("r100",
            100e3), r1000("r1000", 1e6),amp1("amp1"), amp10("amp10"),
            amp100("amp100"), amp1000("amp1000"),
            rswt1("rswt1", 0.0, 1e15, 0), rswt10("rswt10", 0.0, 1e15, 0),
            rswt100("rswt100", 0.0, 1e15, 0),
            rswt1000("rswt1000", 0.0, 1e15, 0)
      {
            //signal conversion
            convs.inp(vin);
            convs.p(n1);
            convs.n(gnd);

            r.n(n1);
            r.p(n2);

            //switches
            rswt1.p(n2);
            rswt1.ctrl(amp1);
            rswt1.n(n3);
            r1.n(n3);   r1.p(n7);

            rswt10.p(n2);
            rswt10.ctrl(amp10);
            rswt10.n(n4);
            r10.n(n4);  r10.p(n7);
```

```
            rswt100.p(n2);
            rswt100.ctrl(amp100);
            rswt100.n(n5);
            r100.n(n5); r100.p(n7);

            rswt1000.p(n2);
            rswt1000.ctrl(amp1000);
            rswt1000.n(n6);
            r1000.n(n6);      r1000.p(n7);

            opamp1.nip(gnd);
            opamp1.nin(n2);
            opamp1.nop(n7);
            opamp1.non(gnd);

            rc.n(n7);
            rc.p(n8);

            opamp2.nip(gnd);
            opamp2.nin(n8);
            opamp2.nop(vout);
            opamp2.non(gnd);

            rf.n(n8);
            rf.p(vout);
      }

private:
      sca_eln::sca_node n1, n2, n3, n4, n5, n6, n7, n8;
      sca_eln::sca_node_ref gnd;
};
```

A.11: DC Shift Model Code

```
#include "systemc.h"
#include "systemc-ams.h"

SC_MODULE(shft)
{
      //Module ports
      sca_tdf::sca_in<double> dcin;
      sca_eln::sca_terminal   sigin;
      sca_eln::sca_terminal   sigout;
      sca_eln::sca_r    r1, r2, r3, r4; //Resistors

      sca_eln::sca_tdf_vsource      shfttdf2eln; //Converter
      sca_eln::sca_nullor     opamp1; //Operational amplifier

      //Module constructor
      shft(sc_core::sc_module_name)
            : dcin("dcin"), sigin("sigin"), sigout("sigout"),
            shfttdf2eln("shfttdf2eln"), opamp1("opamp1"),
            r1("r1", 1e3), r2("r2", 1e3), r3("r3", 2e3), r4("r4", 2e3)
```

```
      {
            //components connections
            r1.n(sigin);      r1.p(n1);

            opamp1.nip(n1);
            opamp1.nin(n3);
            opamp1.nop(sigout);
            opamp1.non(gnd);

            shfttdf2eln.inp(dcin);
            shfttdf2eln.p(n4);
            shfttdf2eln.n(gnd);
            r2.n(n4);   r2.p(n1);

            r4.n(gnd);  r4.p(n3);
            r3.n(n3);   r3.p(sigout);
      }
public:
      sca_eln::sca_node n1, n3, n4;
      sca_eln::sca_node_ref   gnd;
};
```

A.12: Sample and Hold Implementation Model Code

```
#include "systemc.h"
#include "systemc-ams.h"

SC_MODULE(samphold)
{
      //Module ports
      sca_eln::sca_terminal         vin;
      sca_tdf::sca_out<double>      vout;
      sca_tdf::sca_in<bool>         pulsin;

      sca_eln::sca_c    holdcap; //ELN capacitor
      sca_eln::sca_tdf::sca_rswitch swth; //Switch

      sca_eln::sca_nullor     opamp1, opamp2; //Operational amplifiers

      sca_eln::sca_tdf_vsink  conv1; //Converter port

      //Module constructor
      SC_CTOR(samphold)
            : vin("vin"), vout("vout"), opamp1("opamp1"), opamp2("opamp2"),
            gnd("gnd"), conv1("conv1"),
            holdcap("holdcap", 1e1, 0.0),
            swth("swth", 0.0, 1e15, 0)
      {
            //omponent connections
            opamp1.nip(vin);
            opamp1.nin(n1);
            opamp1.nop(n1);
            opamp1.non(gnd);
```

```
            swth.p(n1);
            swth.ctrl(pulsin);
            swth.n(n2);

            holdcap.p(n2);
            holdcap.n(gnd);

            opamp2.nip(n2);
            opamp2.nin(n3);
            opamp2.nop(n3);
            opamp2.non(gnd);

            conv1.p(n3);
            conv1.n(gnd);
            conv1.outp(vout);
      }
private:
      sca_eln::sca_node n1, n2, n3;
      sca_eln::sca_node_ref   gnd;
};
```

A.13: ADC Module Code

```
#include "systemc.h"
#include "systemc-ams.h"

SCA_TDF_MODULE(theadc)
{
      //Module ports
      sca_tdf::sca_in<double> anlgsigin;
      sca_tdf::sca_in<bool>   pulsin;
      //TDF-DE output port
      sca_tdf::sca_de::sca_out<sc_uint<8>>      adcout8;
      //variable declarations
      sc_uint<8>        adcval8;
      float samp;
      int   intsampval8, i8, r8, dev8;
      //Module constructor
      theadc(sc_core::sc_module_name)
            : anlgsigin("anlgsigin"),
            adcout8("adcout8"),
            pulsin("pulsin")
      {}
      void processing()
      {
            if (pulsin.read() == 1)
            {
                  //Sampling
                  samp = (anlgsigin.read());
                  intsampval8 = (int) (samp * 256 / 6);
                  //Conversion
                  for(i8=0; i8<8; i8++)
                  {
                        r8 = intsampval8 % 2;
```

```
                        dev8 = intsampval8 / 2;
                        adcval8[i8] = r8;
                        intsampval8 = dev8;
                  }

                  //Output the digital values
                  adcout8.write(adcval8);
            }
      }
};
```

A.14: Oscilloscope Implementation model Test Bench Code

```
//Include modules
#include "driver.h"
#include "signalsource.cpp"
#include "buffer.cpp"
#include "modulator.cpp"
#include "shifter.cpp"
#include "controlunit.cpp"
#include "ksource.cpp"
#include "pulsesource.cpp"
#include "sampleandhold.cpp"
#include "adc.cpp"

int sc_main(int argc, char *argv[])
{
      sc_report_handler::set_actions("/IEEE_Std_1666/deprecated",
SC_DO_NOTHING);

      //decaring signals for module connection
      sc_signal<double>             t_frequency;
      sc_signal<double>             t_amp;
      sc_signal<double>             t_enbl_ctrl;
      sca_tdf::sca_signal<double>   t_enbl_k;
      sca_tdf::sca_signal<double>   t_enbl_pls;
      sca_tdf::sca_signal<bool>     t_pls;
      sca_tdf::sca_signal<bool>     t_att, t_att10, t_att100;
      sca_tdf::sca_signal<bool>     t_amp1, t_amp10, t_amp100, t_amp1000;
      sca_tdf::sca_signal<double>   t_sine_out;
      sca_tdf::sca_signal<double>   t_buff_sig;
      sca_tdf::sca_signal<double>   t_k_val;
      sca_tdf::sca_signal<double>   t_shftsig;
      sc_signal<sc_uint<8>>         adcsign_8;
      sca_eln::sca_node             t_modsignal;
      sca_eln::sca_node             t_shftsignal;
      sca_tdf::sca_signal<double>   t_SHsignal;

      //Setting system clock
      const sc_time t_period(10, SC_PS);
      sc_clock clk("clk", t_period);

      //Driver module
      syst_driver dr("Driver_Module");
```

```
dr.d_frequency(t_frequency);
dr.d_ampl(t_amp);
dr.clock(clk);
dr.enbl_out(t_enbl_ctrl);

//Control unit module
ctrl ct("Control_Unit");
ct.infrq(t_frequency);
ct.enabl(t_enbl_ctrl);
ct.enb_k(t_enbl_k);
ct.enb_pls_gen(t_enbl_pls);
ct.k_val(t_k);

//Signal source module
sinewv sr("Signal_source");
sr.inpfrq(t_frequency);
sr.inpamp(t_amp);
sr.att10(t_att10);
sr.att100(t_att100);
sr.amp1(t_amp1);
sr.amp10(t_amp10);
sr.amp100(t_amp100);
sr.amp1000(t_amp1000);
sr.out(t_buff_sig);
sr.shiftsig(t_shftsig);

//Buffer module
buff b("Buffer");
b.sigin(t_buff_sig);
b.swtch10(t_att10);
b.swtch100(t_att100);
b.sigout(t_sine_out);

//K value source module
kvals ks("K_source");
ks.enbl_in(t_enbl_k);
ks.frein(t_sine_out);
ks.k_val(t_k_val);

//Modulator module
modgain md("Modulator");
md.vin(t_sine_out);
md.amp1(t_amp1);
md.amp10(t_amp10);
md.amp100(t_amp100);
md.amp1000(t_amp1000);
md.vout(t_modsignal);

//Shifter module
shft sh("Shifter");
sh.dcin(t_shftsig);
sh.sigin(t_modsignal);
sh.sigout(t_shftsignal);
```

```
      //Pulse generator module
      pulses p("Pulse_Generator");
      p.enb(t_enbl_pls);
      p.inpfrq(t_frequency);
      p.pulsout(t_pls);

      //Sample and Hold module
      samphold sp("SampleAndHold");
      sp.vin(t_shftsignal);
      sp.vout(t_SHsignal);
      sp.pulsin(t_pls);

      //ADC module
      theadc dc("ADC");
      dc.pulsin(t_pls);
      dc.anlgsigin(t_SHsignal);
      dc.adcout8(adcsign_8);

      // Tracing waveforms
      sca_util::sca_trace_file *anatf =
sca_util::sca_create_vcd_trace_file("WaveForms");

      cout << "Start tracing waveforms " << endl;
      sca_util::sca_trace(anatf, t_buff_sig, "S(t)");
      sca_util::sca_trace(anatf, t_modsignal, "KS(t)_Mod_out");
      sca_util::sca_trace(anatf, t_pls, "Pulses");
      sca_util::sca_trace(anatf, t_SHsignal ,"KS(t)H(nT)");
      sca_util::sca_trace(anatf, adcsign_8, "D8(n)");
      sca_util::sca_trace(anatf, t_att10, "Attenuate_by_10");
      sca_util::sca_trace(anatf, t_att100, "Attenuate_by_100");
      sca_util::sca_trace(anatf, t_amp1, "Amplify_by_1");
      sca_util::sca_trace(anatf, t_amp10, "Amplify_by_10");
      sca_util::sca_trace(anatf, t_amp100, "Amplify_by_100");
      sca_util::sca_trace(anatf, t_amp1000, "Amplify_by_1000");

      //Simulation period
      sc_start(1, SC_US);

      sca_util::sca_close_vcd_trace_file(anatf);
      cout << "Finished tracing waveforms" << endl;

      return (0);
}
```

APPENDIX B: SIGNAL GENERATOR CODES

B.1: Digital Signal Generator Module Code

```
//Include header files
#include "systemc-ams.h"
#include "systemc.h"
#include "math.h"

SCA_TDF_MODULE(siggen)
{
      //Module ports
      sca_tdf::sc_in<double>        inpfrq;
      sca_tdf::sca_in<double>       theclock;
      sca_tdf::sc_out< sc_uint<8> > sineout;
      sca_tdf::sc_out< sc_uint<8> >       swthout;
      sca_tdf::sc_out< sc_uint<8> >       trgout;
      sca_tdf::sca_out<bool>              sqrout;

      //Module constructor
      siggen(sc_core::sc_module_name nm, double swthval_ = 0.0,
      double trgval_ = 0.0, double myclkT_ = 0.00, double i_ = 0,
      int itr_ = 0, int j_ = 0, int x_ = 0, int  cycls_ = 0,
      sca_core::sca_time tm_ = sca_core::sca_time(10.0, sc_core::SC_NS))
            : sineout("sineout"), inpfrq("inpfrq"), trgout("trgout"),
            sqrout("sqrout"), tm(tm_),swthout("swthout"),
            theclock("theclock"),i(i_), itr(itr_), j(j_), x(x_),
            cycls(cycls_), myclkT(myclkT_)
      {}

      void set_attributes()
      {
            set_timestep(tm);
      }

      void processing()
      {
            //process done when pulse is high
            if (theclock.read() == 1)
            {
                  //Generate signals
                  int sampls, samplstr;
                  double trgval1;
                  double freq = (inpfrq.read());
                  s = 20;
                  sf = s / 2;
                  sampls = 1 / ((20e-9) * freq);
                  myclkT = 1 / (10 * freq);
                  samplstr = sampls / 2;
                  double t = get_time().to_seconds();
                  sinval = (100 + (100 * std::sin(2.0 * 3.142 * freq * t)));
                  cycls = cycls + 1;

                  //saw tooth wave
```

```
if (i <= s)
{
      swthval = i * 100 / s;
      i = i + 1;
}
else
{
      swthval = 0.0;
      i = 0;
      x = x + 1;
      if (x == 4)
      {
            i_time = get_time().to_seconds();
            cycls = 0;
      }
      if (x == 14)
      {
            f_time = get_time().to_seconds();

            time_diff = f_time - i_time;
            avtimepc = time_diff / cycls;
            int y = x - 4;
            double avtmpercycl = ((time_diff) / y);
            double totalsampls = (sampls * 4);
            double instpercycle = ((samplstr * 117) +
            (samplstr * 121));
            double tmprinst = avtmpercycl / instpercycle;
            double oprtgfrq = 1 / tmprinst;
      }
}
//Triangle and square waves
if (itr <= sf)
{
      trgval = itr * 100 / sf;
      sqrout.write(1);
      trgvals[itr] = trgval;
      itr = itr + 1;
      j = j + 1;
}
else
{
      if (itr <= s)
      {
            trgval = trgvals[j - 1];
            sqrout.write(0);  //square wave
            itr = itr + 1;
            j = j - 1;
      }
      else
      {
            trgval = 0;
            itr = 0;
            j = 0;
      }
}
```

```
                    //generate digital signals
                    for (int i = 0; i < 8; i++)
                    {
                          //sinewave conversion
                          rsn = ((int)sinval) % 2;
                          dsn = (int)sinval / 2;
                          binsinval[i] = rsn;
                          sinval = dsn;

                          //sawtooth conversion
                          rsw = ((int)swthval) % 2;
                          dsw = (int)swthval / 2;
                          binswval[i] = rsw;
                          swthval = dsw;

                          //triangle conversion
                          rtg = ((int)trgval) % 2;
                          dtg = (int)trgval / 2;
                          bintgval[i] = rtg;
                          trgval = dtg;
                    }
                    //output signals
                    sineout.write(binsinval);
                    swthout.write(binswval);
                    trgout.write(bintgval);
              }
      }

private:
      double      swthval, trgval, trgvals[1000];
      double      sinval, swval, tgval;
      double i, x;
      double i_time, f_time, time_diff;
      double frqval, avtimepc;
      int j, itr;
      int rsn, dsn;
      int rsw, dsw;
      int rtg, dtg;
      int cycls, s, sf;
      sc_uint<8> binsinval, binswval, bintgval;
      sca_core::sca_time tm;
      double      myclkT, myclk;
};
```

B.2: DAC Functional Model Code

```
#include "systemc-ams.h"
#include "systemc.h"
#include "math.h"

SCA_TDF_MODULE (dac)
{
      //Module ports
```

```
      sca_tdf::sc_in< sc_uint<8> >  sine_in;
      sca_tdf::sc_in< sc_uint<8> >  saw_in;
      sca_tdf::sc_in< sc_uint<8> >  triangle_in;
      sca_tdf::sca_out<double>      sine_out;
      sca_tdf::sca_out<double>      saw_out;
      sca_tdf::sca_out<double>      triangle_out;

      //module constructor
      dac(sc_core::sc_module_name nm,
      double sval_ = 0.0, double swval_ = 0.0, double tgval_ = 0.0,
      sca_core::sca_time tm_ = sca_core::sca_time(20.0, sc_core::SC_NS))
      :     sine_in("sine_in"), saw_in("saw_in"), triangle_in("triangle_in"),
            sine_out("sine_out"), saw_out("saw_out"),
            triangle_out("triangle_out"),
            sval(sval_), swval(swval_), tgval(tgval_),
            tm(tm_)
      {}
      void set_attributes()
      {
            set_timestep(tm);
      }
      void processing()
      {
            //convert signals
            sinval = sine_in.read();
            sawval = saw_in.read();
            trgval = triangle_in.read();
            for (int i = 0; i<8; i++)
            {
                  sval = sval + (sinval[i]* (pow(2,i)));
                  swval = swval + (sawval[i]* (pow(2,i)));
                  tgval = tgval + (trgval[i]* (pow(2,i)));
            }
            //Output signals
            sine_out.write(sval);
            saw_out.write(swval);
            triangle_out.write(tgval);
            sval = 0;
            swval = 0;
            tgval = 0;
      }
private:
      double sval, swval, tgval;
      sc_uint<8> sinval, sawval, trgval;
      sca_core::sca_time tm;
};
```

B.3: Signal Generator Functional Model Test Bench Code

```
//Include modules
#include "systemc-ams.h"
#include "thedriver.h"
#include "theClkGen.cpp"
#include "signalgen.cpp"
```

```
#include "theDAC.cpp"

int sc_main(int argc, char *argv[])
{
      sc_report_handler::set_actions("/IEEE_Std_1666/deprecated",
SC_DO_NOTHING);

      //define signals for module connection
      sc_signal<double>             t_frequency;
      sca_tdf::sca_signal<double>   t_sine;
      sca_tdf::sca_signal<double>   t_swth;
      sca_tdf::sca_signal<double>   t_trgl;
      sca_tdf::sca_signal<bool>     t_sqr;
      sc_core::sc_signal<double>    t_desine;
      sc_core::sc_signal<double>    t_deswth;
      sc_core::sc_signal<double>    t_detrgl;
      sca_tdf::sca_signal<double>   t_sampclk;
      sc_signal<sc_uint<8> >        t_dgsine;
      sc_signal<sc_uint<8> >        t_dgswth;
      sc_signal<sc_uint<8> >        t_dgtgth;
      sca_tdf::sca_signal<double>   t_anasine;
      sca_tdf::sca_signal<double>   t_anaswth;
      sca_tdf::sca_signal<double>   t_anatrgl;

      // declare a time constant for the system clock
      const sc_time t_period(10, SC_NS);
      sc_clock clk("clk", t_period);

      //Driver module
      syst_driver dr("Driver_Module");
      dr.d_frequency(t_frequency);
      dr.clock(clk);

      //Sampling clock generator module
      sampclk scl("SamplingClock");
      scl.d_frequency(t_frequency);
      scl.spclk(t_sampclk);

      //Signal generator module
      siggen sg("SignalGen");
      sg.inpfrq(t_frequency);
      sg.theclock(t_sampclk);
      sg.sineout(t_dgsine);
      sg.swthout(t_dgswth);
      sg.trgout(t_dgtgth);
      sg.sqrout(t_sqr);

      //DAC module
      theDAC dc("DAC");
      dc.sine_in(t_dgsine);
      dc.saw_in(t_dgswth);
      dc.triangle_in(t_dgtgth);
      dc.sine_out(t_anasine);
      dc.saw_out(t_anaswth);
      dc.triangle_out(t_anatrgl);
```

```
      // Tracing waveforms
      sca_util::sca_trace_file *anatf =
sca_util::sca_create_vcd_trace_file("WaveFormsFile");

      cout << "Start tracing waveforms " << endl;
      sca_util::sca_trace(anatf, clk, "SystemClock");
      sca_util::sca_trace(anatf, t_sampclk, "SamplingClock");
      sca_util::sca_trace(anatf, t_sqr, "Square");
      sca_util::sca_trace(anatf, t_anasine, "S(t)_Sine");
      sca_util::sca_trace(anatf, t_anaswth, "S(t)_Sawtooth");
      sca_util::sca_trace(anatf, t_anatrgl, "S(t)_Triangle");

      //Simulation period
      sc_start(250, SC_MS);

      //sc_close_vcd_trace_file (digtf);
      sca_util::sca_close_vcd_trace_file(anatf);
      cout << "Finished tracing waveforms\a\a" << endl;

      return (0);
}
```

B.4: Phase Increment Value Generator module code

```
#include "the_phase_increment.h"
#include "math.h"

void calc_pinc::prc_calc_pinc()
{
      //Variable declaration
      Double      frequency_var;
      double      pinc_var;
      double      val1, val2, val3;

      frequency_var = frequency.read();

      val1 = frequency_var / 200000000;//200000000 is the sampling frequency
      val2 = val1 * 268435456;      //268435456 = 2^28
      val3 = val2 + 0.5;
      pinc_var = val3;
      pinc.write(pinc_var); //phase value output
}
```

B.5: Phase Value Accumulator module code

```
#include "the_phase_accumulator.h"
#include <iostream>
void ph_acc::prc_phase_accumulation()
{
      sc_uint<28> x;   // used to avoid writing to port before if statement
      x = ((ph_acc_reg.read()) + (pinc.read()));
      ph_acc_reg.write(x);
}
```

B.6: Generator module code

```
#include "systemc-ams.h"
#include "the_digital_signal_generator.h"
void ph_to_amplitude::prc_ph_to_amplitude()
{
      //Look-Up-Table
      sc_uint<32> amplitude[256] =
      {
            100,  102,  105,  107,  110,  112,  115,  117,
            120,  122,  124,  127,  129,  131,  134,  136,
            138,  141,  143,  145,  147,  149,  151,  153,
            156,  158,  160,  162,  163,  165,  167,  169,
            171,  172,  174,  176,  177,  179,  180,  182,
            183,  184,  186,  187,  188,  189,  190,  191,
            192,  193,  194,  195,  196,  196,  197,  198,
            198,  199,  199,  199,  200,  200,  200,  200,
            200,  200,  200,  200,  200,  199,  199,  199,
            198,  198,  197,  196,  196,  195,  194,  193,
            192,  191,  190,  189,  188,  187,  186,  184,
            183,  182,  180,  179,  177,  176,  174,  172,
            171,  169,  167,  165,  163,  162,  160,  158,
            156,  153,  151,  149,  147,  145,  143,  141,
            138,  136,  134,  131,  129,  127,  124,  122,
            120,  117,  115,  112,  110,  107,  105,  102,
            100,  98,   95,   93,   90,   88,   85,   83,
            80,   78,   76,   73,   71,   69,   66,   64,
            62,   59,   57,   55,   53,   51,   49,   47,
            44,   42,   40,   38,   37,   35,   33,   31,
            29,   28,   26,   24,   23,   21,   20,   18,
            17,   16,   14,   13,   12,   11,   10,   9,
            8,    7,    6,    5,    4,    4,    3,    2,
            2,    1,    1,    1,    0,    0,    0,    0,
            0,    0,    0,    0,    0,    1,    1,    1,
            2,    2,    3,    4,    4,    5,    6,    7,
            8,    9,    10,   11,   12,   13,   14,   16,
            17,   18,   20,   21,   23,   24,   26,   28,
            29,   31,   33,   35,   37,   38,   40,   42,
            44,   47,   49,   51,   53,   55,   57,   59,
            62,   64,   66,   69,   71,   73,   76,   78,
            80,   83,   85,   88,   90,   93,   95,   98,
      };

      sc_uint<28> quantized_ph_acc_var;
      sc_uint<8> sine_amp_out;
      sc_uint<8>  ph_acc;
      sc_uint<8>  ph_acc1;
      quantized_ph_acc_var = ph_acc_reg.read();
      ph_acc1 = quantized_ph_acc_var >> 20;     //shifts to get the 8 MSB
      sine_amp_out = (amplitude[ph_acc1]);
      sine_out8 = (sine_amp_out);   //sine wave output

}
```

B.7: DAC Non-Functional Model Code

```
#include "systemc-ams.h"
#include "systemc.h"
#include "math.h"

SCA_TDF_MODULE (dac)
{
      //Module ports
      sca_tdf::sc_in< sc_uint<8> >  sine_in;
      sca_tdf::sca_out<double>      sine_out;

      //Module Constructor
      dac(sc_core::sc_module_name nm,
      double sval_ = 0.0, double swval_ = 0.0,
      sca_core::sca_time tm_ = sca_core::sca_time(20.0, sc_core::SC_NS))
      :     sine_in("sine_in"),sine_out("sine_out"),
            sval(sval_), swval(swval_), tm(tm_)
      {}
      void set_attributes()
      {
            set_timestep(tm);
      }
      void processing()
      {
            //convert waves
            sinval = sine_in.read();
            for (int i = 0; i<8; i++)
            {
                  sval = sval + (sinval[i]* (pow(2,i)));
            }
            sine_out.write(sval);
            sval = 0;
      }
private:
      double sval;
      sc_uint<8> sinval;
      sca_core::sca_time tm;
};
```

B.8: Signal Generator None-Functional Test Bench Code

```
//Include modules
#include "systemc-ams.h"
#include "thedriver.h"
#include "theSamplingClockGen.cpp"
#include"the_phase_accumulator.h"
#include"the_phase_increment.h"
#include "the_digital_signal_generator.h"
#include "theDAC.cpp"

int sc_main(int argc, char *argv[])
{
      sc_report_handler::set_actions("/IEEE_Std_1666/deprecated",
SC_DO_NOTHING);
```

```
      //Declaration of signals
      sc_signal<sc_uint<28> > t_frequency;
      sc_signal<double>       t_amp, t_sel, t_tmr;
      sc_signal<sc_uint<28> > t_phsincr;
      sc_signal<sc_uint<28> > t_ph_acc_reg;
      sc_signal<sc_uint<8> >  t_sine_out;
      sc_signal<bool>               t_sampclk;

      //System clock definition
      const sc_time t_period(10, SC_NS);
      sc_clock clk("clk", t_period);

      //Driver module
      syst_driver dr("Driver_Module");
      dr.d_frequency(t_frequency);
      dr.d_ampl(t_amp);
      dr.d_signsel(t_sel);
      dr.clock(clk);

      //Sampling clock generator module
      samp_clk sclk("SamplingClock");
      sclk.clock(clk);
      sclk.d_frequency(t_frequency);
      sclk.d_sampclk(t_sampclk);

      //Phase increment module
      calc_pinc phsinc("PhaseIncrement");
      phsinc.clock(clk);
      phsinc.frequency(t_frequency);
      phsinc.pinc(t_phsincr);

      //Phase accumulator module
      ph_acc phsacum("PhaseAccumulator");
      phsacum.clock(clk);
      phsacum.pinc(t_phsincr);
      phsacum.ph_acc_reg(t_ph_acc_reg);

      //Phase value to amplitude module
      ph_to_amplitude pamp("Phase_to_Amplitude");
      pamp.ph_acc_reg(t_ph_acc_reg);
      pamp.clock(t_sampclk);
      pamp.sine_out8(t_sine_out);

      //DAC module
      theDAC dc("The_DAC");
      dc.sine_in(t_sine_out);
      dc.ampl_in(t_amp);
      dc.sign_sel(t_sel);
      dc.samp_in(t_sampclk);
      dc.sign_out(t_anlgsig);

      // Tracing waveforms
      sca_util::sca_trace_file *anatf =
sca_util::sca_create_vcd_trace_file("WaveForms");
```

```
	cout << "Start tracing waveforms " << endl;

	sca_util::sca_trace(anatf, t_sampclk, "Samp_Clock");
	sca_util::sca_trace(anatf, t_anlgsig, "S(t)");
	//Simulation period
	sc_start(10, SC_MS);

	sca_util::sca_close_vcd_trace_file(anatf);
	cout << "Finished tracing waveforms\a\a" << endl;

	return (0);
}
```

B.8: DAC Implementation Model Code

```
#include "systemc-ams.h"
#include "systemc.h"

SC_MODULE(dac)
{
	//input terminals
	sca_eln::sca_terminal   swdc0, swdc1, swdc2, swdc3, swdc4, swdc5;
	sca_eln::sca_terminal   swdc6, swdc7;
	sca_eln::sca_terminal   stdc0, stdc1, stdc2, stdc3, stdc4, stdc5;
	sca_eln::sca_terminal   stdc6, stdc7;
	sca_eln::sca_terminal   tgdc0, tgdc1, tgdc2, tgdc3, tgdc4, tgdc5;
	sca_eln::sca_terminal   tgdc6, tgdc7;

	//output terminals
	sca_eln::sca_terminal   swdacout;
	sca_eln::sca_terminal   stdacout;
	sca_eln::sca_terminal   tgdacout;

	//Resistor declarations
	sca_eln::sca_r    swr10, swr11, swr12, swr13, swr14, swr15, swr16;
	sca_eln::sca_r    swr17;
	sca_eln::sca_r    str10, str11, str12, str13, str14, str15, str16;
	sca_eln::sca_r    str17;
	sca_eln::sca_r    tgr10, tgr11, tgr12, tgr13, tgr14, tgr15, tgr16;
	sca_eln::sca_r    tgr17;
	sca_eln::sca_r    swr2in, swr20, swr21, swr22, swr23, swr24, swr25;
	sca_eln::sca_r    swr26, swr27, swr2f;
	sca_eln::sca_r    str2in, str20, str21, str22, str23, str24, str25;
	sca_eln::sca_r    str26, str27, str2f;
	sca_eln::sca_r    tgr2in, tgr20, tgr21, tgr22, tgr23, tgr24, tgr25;
	sca_eln::sca_r    tgr26, tgr27, tgr2f;

	//Operational Amplifier declarations
	sca_eln::sca_nullor     swnull1, stnull1, tgnull1;

	//Module Constructor connections
	SC_CTOR(dac)
		: swdc0("sw5V_input0"), swdc1("sw5V_input1"),
		swdc2("sw5V_input2"), swdc3("sw5V_input3"),
```

```
swdc4("sw5V_input4"),swdc5("sw5V_input5"),
swdc6("sw5V_input6"),swdc7("sw5V_input7"),
swdacout("swsinewaveput_terminal"),

stdc0("st5V_input0"), stdc1("st5V_input1"), stdc2("st5V_input2"),
stdc3("st5V_input3"), stdc4("st5V_input4"),stdc5("st5V_input5"),
stdc6("st5V_input6"),stdc7("st5V_input7"),
stdacout("stsinewaveput_terminal"),

tgdc0("tg5V_input0"), tgdc1("tg5V_input1"), tgdc2("tg5V_input2"),
tgdc3("tg5V_input3"), tgdc4("tg5V_input4"), tgdc5("tg5V_input5"),
tgdc6("tg5V_input6"),tgdc7("tg5V_input7"),
tgdacout("tgsinewaveput_terminal"),

swr10("swr10", 10.0e3), swr11("swr11", 10.0e3), swr12("swr12",
10.0e3), swr13("swr13", 10.0e3), swr14("swr14", 10.0e3),
swr15("swr15", 10.0e3),swr16("swr61", 10.0e3), swr17("swr17",
10.0e3),

str10("str10", 10.0e3), str11("str11", 10.0e3), str12("str12",
10.0e3), str13("str13", 10.0e3), str14("str14", 10.0e3),
str15("str15", 10.0e3), str16("str61", 10.0e3), str17("str17",
10.0e3),

tgr10("tgr10", 10.0e3), tgr11("tgr11", 10.0e3), tgr12("tgr12",
10.0e3), tgr13("tgr13", 10.0e3), tgr14("tgr14", 10.0e3),
tgr15("tgr15", 10.0e3),tgr16("tgr61", 10.0e3), tgr17("tgr17",
10.0e3),

swr2in("swr2in", 20.0e3), swr20("swr20", 20.0e3), swr21("swr21",
20.0e3), swr22("swr22", 20.0e3), swr23("swr23", 20.0e3),
swr24("swr24", 20.0e3), swr25("swr25", 20.0e3),
swr26("swr26", 20.0e3), swr27("swr27", 20.0e3), swr2f("swr2f",
-20.0e3),

str2in("str2in", 20.0e3), str20("str20", 20.0e3), str21("str21",
20.0e3), str22("str22", 20.0e3), str23("str23", 20.0e3),
str24("str24", 20.0e3), str25("str25", 20.0e3),
str26("str26", 20.0e3), str27("str27", 20.0e3), str2f("str2f", -
20.0e3),

tgr2in("tgr2in", 20.0e3), tgr20("tgr20", 20.0e3), tgr21("tgr21",
20.0e3), tgr22("tgr22", 20.0e3), tgr23("tgr23", 20.0e3),
tgr24("tgr24", 20.0e3), tgr25("tgr25", 20.0e3),
tgr26("tgr26", 20.0e3), tgr27("tgr27", 20.0e3), tgr2f("tgr2f", -
20.0e3),

swnull1("swNullator"), stnull1("stNullator"),
tgnull1("tgNullator"),swn("swn"), swn0("swn0"), swn1("swn1"),
swn2("swn2"), swn3("swn3"), swn4("swn4"), swn5("swn5"),
swn6("swn6"), swn7("swn7"), stn("stn"), stn0("stn0"),
stn1("stn1"), stn2("stn2"), stn3("stn3"), stn4("stn4"),
stn5("stn5"), stn6("stn6"), stn7("stn7"),tgn("tgn"),
tgn0("tgn0"), tgn1("tgn1"), tgn2("tgn2"), tgn3("tgn3"),
tgn4("tgn4"), tgn5("tgn5"), tgn6("tgn6"), tgn7("tgn7"),
```

```
        gnd("Ground")
{
        //Component connections
        swr2in.n(gnd);    swr10.n(swn);     swr11.n(swn0);
        swr12.n(swn1);    swr13.n(swn2);    swr14.n(swn3);
        swr15.n(swn4);    swr16.n(swn5);    swr17.n(swn6);
        swr2in.p(swn);    swr10.p(swn0);    swr11.p(swn1);
        swr12.p(swn2);    swr13.p(swn3);    swr14.p(swn4);
        swr15.p(swn5);    swr16.p(swn6);    swr17.p(swn7);

        swr2f.n(swn7);
        swr2f.p(swdacout);

        swr20.p(swn0);    swr21.p(swn1);    swr22.p(swn2);
        swr23.p(swn3);    swr24.p(swn4);    swr25.p(swn5);
        swr26.p(swn6);    swr27.p(swn7);    swr20.n(swdc0);
        swr21.n(swdc1);   swr22.n(swdc2);   swr23.n(swdc3);
        swr24.n(swdc4);   swr25.n(swdc5);   swr26.n(swdc6);
        swr27.n(swdc7);

        //Op Amp null1 connections
        swnull1.nip(swn7);
        swnull1.nin(gnd);
        swnull1.nop(swdacout);
        swnull1.non(gnd);

        str2in.n(gnd);    str10.n(stn);     str11.n(stn0);
        str12.n(stn1);    str13.n(stn2);    str14.n(stn3);
        str15.n(stn4);    str16.n(stn5);    str17.n(stn6);
        str2in.p(stn);    str10.p(stn0);    str11.p(stn1);
        str12.p(stn2);    str13.p(stn3);    str14.p(stn4);
        str15.p(stn5);    str16.p(stn6);    str17.p(stn7);

        str2f.n(stn7);
        str2f.p(stdacout);

        str20.p(stn0);    str21.p(stn1);    str22.p(stn2);
        str23.p(stn3);    str24.p(stn4);    str25.p(stn5);
        str26.p(stn6);    str27.p(stn7);
        str20.n(stdc0);   str21.n(stdc1);   str22.n(stdc2);
        str23.n(stdc3);   str24.n(stdc4);   str25.n(stdc5);
        str26.n(stdc6);   str27.n(stdc7);

        //Op Amp null1 connections
        stnull1.nip(stn7);
        stnull1.nin(gnd);
        stnull1.nop(stdacout);
        stnull1.non(gnd);

        tgr2in.n(gnd);    tgr10.n(tgn);     tgr11.n(tgn0);
        tgr12.n(tgn1);    tgr13.n(tgn2);    tgr14.n(tgn3);
        tgr15.n(tgn4);    tgr16.n(tgn5);    tgr17.n(tgn6);
        tgr2in.p(tgn);    tgr10.p(tgn0);    tgr11.p(tgn1);
        tgr12.p(tgn2);    tgr13.p(tgn3);    tgr14.p(tgn4);
        tgr15.p(tgn5);    tgr16.p(tgn6);    tgr17.p(tgn7);
```

```
            tgr2f.n(tgn7);
            tgr2f.p(tgdacout);

            tgr20.p(tgn0);    tgr21.p(tgn1);    tgr22.p(tgn2);
            tgr23.p(tgn3);    tgr24.p(tgn4);    tgr25.p(tgn5);
            tgr26.p(tgn6);    tgr27.p(tgn7);
            tgr20.n(tgdc0);   tgr21.n(tgdc1);   tgr22.n(tgdc2);
            tgr23.n(tgdc3);   tgr24.n(tgdc4);   tgr25.n(tgdc5);
            tgr26.n(tgdc6);   tgr27.n(tgdc7);

            //Op Amp null1 connections
            tgnull1.nip(tgn7);
            tgnull1.nin(gnd);
            tgnull1.nop(tgdacout);
            tgnull1.non(gnd);
      }
private:
      sca_eln::sca_node swn, swn0, swn1, swn2, swn3, swn4, swn5, swn6, swn7;
      sca_eln::sca_node stn, stn0, stn1, stn2, stn3, stn4, stn5, stn6, stn7;
      sca_eln::sca_node tgn, tgn0, tgn1, tgn2, tgn3, tgn4, tgn5, tgn6, tgn7;
      sca_eln::sca_node ncon;
      sca_eln::sca_node_ref   gnd;
};
```

B.9: Signal Generator Implementation Model Test Bench Code

```
//Include modules
#include "systemc-ams.h"
#include "thedriver.h"
#include"thephase_accumulator.h"
#include"thephase_increment.h"
#include "theDigital_signal_generator.h"
#include "thedcsource.cpp"
#include "thetdf2lsfconverter.cpp"
#include "thelsf2elnconverter.cpp"
#include "theDACmultiplexer.cpp"
#include "theDAC.cpp"

int sc_main(int argc, char *argv[])
{
      sc_report_handler::set_actions("/IEEE_Std_1666/deprecated",
      SC_DO_NOTHING);

      //Signal declarations
      sc_signal<sc_uint<28> > t_frequency;
      sc_signal<sc_uint<28> > t_phsincr;
      sc_signal<sc_uint<28> > t_ph_acc_reg;
      sc_signal<double>       t_amp;
      sc_signal<bool>         t_square_out;
      sc_signal<sc_uint<8> >  t_saw_out;
      sc_signal<sc_uint<8> >  t_sine_out;
      sc_signal<sc_uint<8> >  t_triangle_out;
      sc_signal<sc_uint<8> >  t_sel_sign_to_DACMUX;
```

```
//Sine wave connecting signals
sca_tdf::sca_signal<double>   t_swtdfdc05, t_swtdfdc15, t_swtdfdc25;
sca_tdf::sca_signal<double>   t_swtdfdc35, t_swtdfdc45, t_swtdfdc55;
sca_tdf::sca_signal<double>   t_swtdfdc65, t_swtdfdc75;
sca_tdf::sca_signal<double>   t_swtdfdc00, t_swtdfdc10, t_swtdfdc20;
sca_tdf::sca_signal<double>   t_swtdfdc30, t_swtdfdc40, t_swtdfdc50;
sca_tdf::sca_signal<double>   t_swtdfdc60, t_swtdfdc70;
sca_lsf::sca_signal<double>   swlsfconvsig00, swlsfconvsig10;
sca_tdf::sca_signal<double>   swlsfconvsig20, swlsfconvsig30;
sca_tdf::sca_signal<double>   swlsfconvsig40, swlsfconvsig50;
sca_tdf::sca_signal<double>   swlsfconvsig60, swlsfconvsig70;
sca_lsf::sca_signal<double>   swlsfconvsig05, swlsfconvsig15;
sca_tdf::sca_signal<double>   swlsfconvsig25, swlsfconvsig35;
sca_tdf::sca_signal<double>   swlsfconvsig45, swlsfconvsig55;
sca_tdf::sca_signal<double>   swlsfconvsig65, swlsfconvsig75;
sca_lsf::sca_signal <double>  swlsfdc0, swlsfdc1, swlsfdc2, swlsfdc3;
sca_lsf::sca_signal <double>  swlsfdc4, swlsfdc5, swlsfdc6, swlsfdc7;
sc_signal<bool>   t_swb0, t_swb1, t_swb2, t_swb3, t_swb4, t_swb5;
sc_signal<bool>   t_swb6, t_swb7;
sca_eln::sca_node       t_sinDACout;
sca_eln::sca_node       t_swelndc0, t_swelndc1, t_swelndc2, t_swelndc3;
sca_eln::sca_node       t_swelndc4, t_swelndc5, t_swelndc6, t_swelndc7;

//SawTooth connecting signals
sca_tdf::sca_signal<double>   t_sttdfdc00, t_sttdfdc10, t_sttdfdc20;
sca_tdf::sca_signal<double>   t_sttdfdc30, t_sttdfdc40, t_sttdfdc5;
sca_tdf::sca_signal<double>   t_sttdfdc60, t_sttdfdc70;
sca_tdf::sca_signal<double>   t_sttdfdc05, t_sttdfdc15, t_sttdfdc25;
sca_tdf::sca_signal<double>   t_sttdfdc35, t_sttdfdc45, t_sttdfdc55;
sca_tdf::sca_signal<double>   t_sttdfdc65, t_sttdfdc75;
sca_lsf::sca_signal     stlsfconvsig00, stlsfconvsig10, stlsfconvsig20;
sca_lsf::sca_signal     stlsfconvsig30, stlsfconvsig40, stlsfconvsig50;
sca_lsf::sca_signal     stlsfconvsig60,stlsfconvsig70;
sca_lsf::sca_signal     stlsfconvsig05, stlsfconvsig15, stlsfconvsig25;
sca_lsf::sca_signal     stlsfconvsig35, stlsfconvsig45, stlsfconvsig55;
sca_lsf::sca_signal     stlsfconvsig65, stlsfconvsig75;
sca_lsf::sca_signal     stlsfdc0, stlsfdc1, stlsfdc2, stlsfdc3;
sca_lsf::sca_signal     stlsfdc4, stlsfdc5, stlsfdc6, stlsfdc7;
sc_signal<bool>   t_stb0, t_stb1, t_stb2, t_stb3, t_stb4, t_stb5;
sc_signal<bool>   t_stb6, t_stb7;
sca_eln::sca_node t_sawthDACout;
sca_eln::sca_node t_stelndc0, t_stelndc1, t_stelndc2, t_stelndc3;
sca_eln::sca_node t_stelndc4, t_stelndc5, t_stelndc6, t_stelndc7;

//Triangle wave connecting signals
sca_tdf::sca_signal<double>   t_tgtdfdc05, t_tgtdfdc15, t_tgtdfdc25;
sca_tdf::sca_signal<double>   t_tgtdfdc35, t_tgtdfdc45, t_tgtdfdc55;
sca_tdf::sca_signal<double>   t_tgtdfdc65, t_tgtdfdc75;
sca_tdf::sca_signal<double>   t_tgtdfdc00, t_tgtdfdc10, t_tgtdfdc20;
sca_tdf::sca_signal<double>   t_tgtdfdc30, t_tgtdfdc40, t_tgtdfdc50;
sca_tdf::sca_signal<double>   t_tgtdfdc60, t_tgtdfdc70;
sca_lsf::sca_signal tglsfconvsig00, tglsfconvsig10, tglsfconvsig20;
sca_lsf::sca_signal tglsfconvsig30, tglsfconvsig40, tglsfconvsig50;
sca_lsf::sca_signal tglsfconvsig60, tglsfconvsig70;
```

```
sca_lsf::sca_signal tglsfconvsig05, tglsfconvsig15, tglsfconvsig25;
sca_lsf::sca_signal tglsfconvsig35, tglsfconvsig45, tglsfconvsig55;
sca_lsf::sca_signal tglsfconvsig65, tglsfconvsig75;
sca_lsf::sca_signal tglsfdc0, tglsfdc1, tglsfdc2, tglsfdc3, tglsfdc4;
sca_lsf::sca_signal tglsfdc5, tglsfdc6, tglsfdc7;
sc_signal<bool>   t_tgb0, t_tgb1, t_tgb2, t_tgb3, t_tgb4, t_tgb5;
sc_signal<bool>   t_tgb6, t_tgb7;
sca_eln::sca_node t_tgDACout;
sca_eln::sca_node t_tgelndc0, t_tgelndc1, t_tgelndc2, t_tgelndc3;
sca_eln::sca_node t_tgelndc4, t_tgelndc5, t_tgelndc6, t_tgelndc7;

//Declaring system clock
const sc_time t_period(20, SC_NS);
sc_clock clk("clk", t_period);

//Driver module
syst_driver dr("Driver_Module");
dr.d_frequency(t_frequency);
dr.clock(clk);

//Phase increment module
calc_pinc phsinc("PhaseIncrement");
phsinc.clock(clk);
phsinc.frequency(t_frequency);
phsinc.pinc(t_phsincr);

//Phase accumulator module
ph_acc phsacum("PhaseAccumulator");
phsacum.clock(clk);
phsacum.pinc(t_phsincr);
phsacum.ph_acc_reg(t_ph_acc_reg);

//Phase value to amplitude module
ph_to_amplitude pamp("Phase_to_Amplitude");
pamp.ph_acc_reg(t_ph_acc_reg);
pamp.clock(clk);
pamp.sine_out(t_sine_out);
pamp.saw_out(t_saw_out);
pamp.triangle_out(t_triangle_out);
pamp.swb0(t_swb0); pamp.swb1(t_swb1); pamp.swb2(t_swb2);
pamp.swb3(t_swb3); pamp.swb4(t_swb4);
pamp.swb5(t_swb5); pamp.swb6(t_swb6); pamp.swb7(t_swb7);
pamp.stb0(t_stb0); pamp.stb1(t_stb1); pamp.stb2(t_stb2);
pamp.stb3(t_stb3); pamp.stb4(t_stb4);
pamp.stb5(t_stb5); pamp.stb6(t_stb6); pamp.stb7(t_stb7);
pamp.tgb0(t_tgb0); pamp.tgb1(t_tgb1); pamp.tgb2(t_tgb2);
pamp.tgb3(t_tgb3); pamp.tgb4(t_tgb4);
pamp.tgb5(t_tgb5); pamp.tgb6(t_tgb6); pamp.tgb7(t_tgb7);

//Square wave binding
pamp.square_out(t_square_out);

//DC source module connections
tgdcsrc tgdc("DC_Source");
tgdc.swout05(t_swtdfdc05);    tgdc.swout00(t_swtdfdc00);
```

```
tgdc.swout15(t_swtdfdc15);    tgdc.swout10(t_swtdfdc10);
tgdc.swout25(t_swtdfdc25);    tgdc.swout20(t_swtdfdc20);
tgdc.swout35(t_swtdfdc35);    tgdc.swout30(t_swtdfdc30);
tgdc.swout45(t_swtdfdc45);    tgdc.swout40(t_swtdfdc40);
tgdc.swout55(t_swtdfdc55);    tgdc.swout50(t_swtdfdc50);
tgdc.swout65(t_swtdfdc65);    tgdc.swout60(t_swtdfdc60);
tgdc.swout75(t_swtdfdc75);    tgdc.swout70(t_swtdfdc70);
tgdc.stout05(t_sttdfdc05);    tgdc.stout00(t_sttdfdc00);
tgdc.stout15(t_sttdfdc15);    tgdc.stout10(t_sttdfdc10);
tgdc.stout25(t_sttdfdc25);    tgdc.stout20(t_sttdfdc20);
tgdc.stout35(t_sttdfdc35);    tgdc.stout30(t_sttdfdc30);
tgdc.stout45(t_sttdfdc45);    tgdc.stout40(t_sttdfdc40);
tgdc.stout55(t_sttdfdc55);    tgdc.stout50(t_sttdfdc50);
tgdc.stout65(t_sttdfdc65);    tgdc.stout60(t_sttdfdc60);
tgdc.stout75(t_sttdfdc75);    tgdc.stout70(t_sttdfdc70);
tgdc.tgout05(t_tgtdfdc05);    tgdc.tgout00(t_tgtdfdc00);
tgdc.tgout15(t_tgtdfdc15);    tgdc.tgout10(t_tgtdfdc10);
tgdc.tgout25(t_tgtdfdc25);    tgdc.tgout20(t_tgtdfdc20);
tgdc.tgout35(t_tgtdfdc35);    tgdc.tgout30(t_tgtdfdc30);
tgdc.tgout45(t_tgtdfdc45);    tgdc.tgout40(t_tgtdfdc40);
tgdc.tgout55(t_tgtdfdc55);    tgdc.tgout50(t_tgtdfdc50);
tgdc.tgout65(t_tgtdfdc65);    tgdc.tgout60(t_tgtdfdc60);
tgdc.tgout75(t_tgtdfdc75);    tgdc.tgout70(t_tgtdfdc70);

//TDF to LSF conversion module connections
tgtdf2lsf tt("TDF2LSF_Converter");
tt.swtdfin00(t_swtdfdc00);    tt.swtdfin10(t_swtdfdc10);
tt.swtdfin20(t_swtdfdc20);    tt.swtdfin30(t_swtdfdc30);
tt.swtdfin40(t_swtdfdc40);    tt.swtdfin50(t_swtdfdc50);
tt.swtdfin60(t_swtdfdc60);    tt.swtdfin70(t_swtdfdc70);
tt.swtdfin05(t_swtdfdc05);    tt.swtdfin15(t_swtdfdc15);
tt.swtdfin25(t_swtdfdc25);    tt.swtdfin35(t_swtdfdc35);
tt.swtdfin45(t_swtdfdc45);    tt.swtdfin55(t_swtdfdc55);
tt.swtdfin65(t_swtdfdc65);    tt.swtdfin75(t_swtdfdc75);
tt.sttdfin00(t_sttdfdc00);    tt.sttdfin10(t_sttdfdc10);
tt.sttdfin20(t_sttdfdc20);    tt.sttdfin30(t_sttdfdc30);
tt.sttdfin40(t_sttdfdc40);    tt.sttdfin50(t_sttdfdc50);
tt.sttdfin60(t_sttdfdc60);    tt.sttdfin70(t_sttdfdc70);
tt.sttdfin05(t_sttdfdc05);    tt.sttdfin15(t_sttdfdc15);
tt.sttdfin25(t_sttdfdc25);    tt.sttdfin35(t_sttdfdc35);
tt.sttdfin45(t_sttdfdc45);    tt.sttdfin55(t_sttdfdc55);
tt.sttdfin65(t_sttdfdc65);    tt.sttdfin75(t_sttdfdc75);
tt.tgtdfin00(t_tgtdfdc00);    tt.tgtdfin10(t_tgtdfdc10);
tt.tgtdfin20(t_tgtdfdc20);    tt.tgtdfin30(t_tgtdfdc30);
tt.tgtdfin40(t_tgtdfdc40);    tt.tgtdfin50(t_tgtdfdc50);
tt.tgtdfin60(t_tgtdfdc60);    tt.tgtdfin70(t_tgtdfdc70);
tt.tgtdfin05(t_tgtdfdc05);    tt.tgtdfin15(t_tgtdfdc15);
tt.tgtdfin25(t_tgtdfdc25);    tt.tgtdfin35(t_tgtdfdc35);
tt.tgtdfin45(t_tgtdfdc45);    tt.tgtdfin55(t_tgtdfdc55);
tt.tgtdfin65(t_tgtdfdc65);    tt.tgtdfin75(t_tgtdfdc75);
tt.swy00(swlsfconvsig00);     tt.swy10(swlsfconvsig10);
tt.swy20(swlsfconvsig20);     tt.swy30(swlsfconvsig30);
tt.swy40(swlsfconvsig40);     tt.swy50(swlsfconvsig50);
tt.swy60(swlsfconvsig60);     tt.swy70(swlsfconvsig70);
tt.swy05(swlsfconvsig05);     tt.swy15(swlsfconvsig15);
```

```
tt.swy25(swlsfconvsig25);     tt.swy35(swlsfconvsig35);
tt.swy45(swlsfconvsig45);     tt.swy55(swlsfconvsig55);
tt.swy65(swlsfconvsig65);     tt.swy75(swlsfconvsig75);
tt.sty00(stlsfconvsig00);     tt.sty10(stlsfconvsig10);
tt.sty20(stlsfconvsig20);     tt.sty30(stlsfconvsig30);
tt.sty40(stlsfconvsig40);     tt.sty50(stlsfconvsig50);
tt.sty60(stlsfconvsig60);     tt.sty70(stlsfconvsig70);
tt.sty05(stlsfconvsig05);     tt.sty15(stlsfconvsig15);
tt.sty25(stlsfconvsig25);     tt.sty35(stlsfconvsig35);
tt.sty45(stlsfconvsig45);     tt.sty55(stlsfconvsig55);
tt.sty65(stlsfconvsig65);     tt.sty75(stlsfconvsig75);
tt.tgy00(tglsfconvsig00);     tt.tgy10(tglsfconvsig10);
tt.tgy20(tglsfconvsig20);     tt.tgy30(tglsfconvsig30);
tt.tgy40(tglsfconvsig40);     tt.tgy50(tglsfconvsig50);
tt.tgy60(tglsfconvsig60);     tt.tgy70(tglsfconvsig70);
tt.tgy05(tglsfconvsig05);     tt.tgy15(tglsfconvsig15);
tt.tgy25(tglsfconvsig25);     tt.tgy35(tglsfconvsig35);
tt.tgy45(tglsfconvsig45);     tt.tgy55(tglsfconvsig55);
tt.tgy65(tglsfconvsig65);     tt.tgy75(tglsfconvsig75);

//DAC multiplexer module connections
DACmux tgx("DAC_Multiplexer");
tgx.swx00(swlsfconvsig00);    tgx.swx10(swlsfconvsig10);
tgx.swx20(swlsfconvsig20);    tgx.swx30(swlsfconvsig30);
tgx.swx40(swlsfconvsig40);    tgx.swx50(swlsfconvsig50);
tgx.swx60(swlsfconvsig60);    tgx.swx70(swlsfconvsig70);
tgx.swx05(swlsfconvsig05);    tgx.swx15(swlsfconvsig15);
tgx.swx25(swlsfconvsig25);    tgx.swx35(swlsfconvsig35);
tgx.swx45(swlsfconvsig45);    tgx.swx55(swlsfconvsig55);
tgx.swx65(swlsfconvsig65);    tgx.swx75(swlsfconvsig75);
tgx.stx00(stlsfconvsig00);    tgx.stx10(stlsfconvsig10);
tgx.stx20(stlsfconvsig20);    tgx.stx30(stlsfconvsig30);
tgx.stx40(stlsfconvsig40);    tgx.stx50(stlsfconvsig50);
tgx.stx60(stlsfconvsig60);    tgx.stx70(stlsfconvsig70);
tgx.stx05(stlsfconvsig05);    tgx.stx15(stlsfconvsig15);
tgx.stx25(stlsfconvsig25);    tgx.stx35(stlsfconvsig35);
tgx.stx45(stlsfconvsig45);    tgx.stx55(stlsfconvsig55);
tgx.stx65(stlsfconvsig65);    tgx.stx75(stlsfconvsig75);
tgx.tgx00(tglsfconvsig00);    tgx.tgx10(tglsfconvsig10);
tgx.tgx20(tglsfconvsig20);    tgx.tgx30(tglsfconvsig30);
tgx.tgx40(tglsfconvsig40);    tgx.tgx50(tglsfconvsig50);
tgx.tgx60(tglsfconvsig60);    tgx.tgx70(tglsfconvsig70);
tgx.tgx05(tglsfconvsig05);    tgx.tgx15(tglsfconvsig15);
tgx.tgx25(tglsfconvsig25);    tgx.tgx35(tglsfconvsig35);
tgx.tgx45(tglsfconvsig45);    tgx.tgx55(tglsfconvsig55);
tgx.tgx65(tglsfconvsig65);    tgx.tgx75(tglsfconvsig75);
tgx.swy0(swlsfdc0);     tgx.swy1(swlsfdc1);
tgx.swy2(swlsfdc2);     tgx.swy3(swlsfdc3);
tgx.swy4(swlsfdc4);     tgx.swy5(swlsfdc5);
tgx.swy6(swlsfdc6);     tgx.swy7(swlsfdc7);
tgx.sty0(stlsfdc0);     tgx.sty1(stlsfdc1);
tgx.sty2(stlsfdc2);     tgx.sty3(stlsfdc3);
tgx.sty4(stlsfdc4);     tgx.sty5(stlsfdc5);
tgx.sty6(stlsfdc6);     tgx.sty7(stlsfdc7);
tgx.tgy0(tglsfdc0);     tgx.tgy1(tglsfdc1);
```

```
tgx.tgy2(tglsfdc2);     tgx.tgy3(tglsfdc3);
tgx.tgy4(tglsfdc4);     tgx.tgy5(tglsfdc5);
tgx.tgy6(tglsfdc6);     tgx.tgy7(tglsfdc7);

tgx.swb0(t_swb0); tgx.swb1(t_swb1); tgx.swb2(t_swb2); tgx.swb3(t_swb3);
tgx.swb4(t_swb4); tgx.swb5(t_swb5); tgx.swb6(t_swb6); tgx.swb7(t_swb7);

tgx.stb0(t_stb0); tgx.stb1(t_stb1); tgx.stb2(t_stb2); tgx.stb3(t_stb3);
tgx.stb4(t_stb4); tgx.stb5(t_stb5); tgx.stb6(t_stb6); tgx.stb7(t_stb7);

tgx.tgb0(t_tgb0); tgx.tgb1(t_tgb1); tgx.tgb2(t_tgb2); tgx.tgb3(t_tgb3);
tgx.tgb4(t_tgb4); tgx.tgb5(t_tgb5); tgx.tgb6(t_tgb6); tgx.tgb7(t_tgb7);

//LSF to ELN conversion module connections
tglsf2eln   tgl("LSF_2_ELN_Converter");
tgl.lsfswin0(swlsfdc0); tgl.lsfswin1(swlsfdc1);
tgl.lsfswin2(swlsfdc2);       tgl.lsfswin3(swlsfdc3);
tgl.lsfswin4(swlsfdc4);       tgl.lsfswin5(swlsfdc5);
tgl.lsfswin6(swlsfdc6);       tgl.lsfswin7(swlsfdc7);
tgl.lsfstin0(stlsfdc0);       tgl.lsfstin1(stlsfdc1);
tgl.lsfstin2(stlsfdc2);       tgl.lsfstin3(stlsfdc3);
tgl.lsfstin4(stlsfdc4);       tgl.lsfstin5(stlsfdc5);
tgl.lsfstin6(stlsfdc6);       tgl.lsfstin7(stlsfdc7);
tgl.lsftgin0(tglsfdc0);       tgl.lsftgin1(tglsfdc1);
tgl.lsftgin2(tglsfdc2);       tgl.lsftgin3(tglsfdc3);
tgl.lsftgin4(tglsfdc4);       tgl.lsftgin5(tglsfdc5);
tgl.lsftgin6(tglsfdc6);       tgl.lsftgin7(tglsfdc7);
tgl.elnswout0(t_swelndc0);    tgl.elnswout1(t_swelndc1);
tgl.elnswout2(t_swelndc2);    tgl.elnswout3(t_swelndc3);
tgl.elnswout4(t_swelndc4);    tgl.elnswout5(t_swelndc5);
tgl.elnswout6(t_swelndc6);    tgl.elnswout7(t_swelndc7);
tgl.elnstout0(t_stelndc0);    tgl.elnstout1(t_stelndc1);
tgl.elnstout2(t_stelndc2);    tgl.elnstout3(t_stelndc3);
tgl.elnstout4(t_stelndc4);    tgl.elnstout5(t_stelndc5);
tgl.elnstout6(t_stelndc6);    tgl.elnstout7(t_stelndc7);
tgl.elntgout0(t_tgelndc0);    tgl.elntgout1(t_tgelndc1);
tgl.elntgout2(t_tgelndc2);    tgl.elntgout3(t_tgelndc3);
tgl.elntgout4(t_tgelndc4);    tgl.elntgout5(t_tgelndc5);
tgl.elntgout6(t_tgelndc6);    tgl.elntgout7(t_tgelndc7);

//DAC module connections
theDAC tdac("The_DAC");
tdac.swdc0(t_swelndc0); tdac.swdc1(t_swelndc1); tdac.swdc2(t_swelndc2);
tdac.swdc3(t_swelndc3); tdac.swdc4(t_swelndc4); tdac.swdc5(t_swelndc5);
tdac.swdc6(t_swelndc6); tdac.swdc7(t_swelndc7);
tdac.swdacout(t_sinDACout);

tdac.stdc0(t_stelndc0); tdac.stdc1(t_stelndc1); tdac.stdc2(t_stelndc2);
tdac.stdc3(t_stelndc3); tdac.stdc4(t_stelndc4);tdac.stdc5(t_stelndc5);
tdac.stdc6(t_stelndc6); tdac.stdc7(t_stelndc7);
tdac.stdacout(t_sawthDACout);

tdac.tgdc0(t_tgelndc0); tdac.tgdc1(t_tgelndc1); tdac.tgdc2(t_tgelndc2);
tdac.tgdc3(t_tgelndc3); tdac.tgdc4(t_tgelndc4); tdac.tgdc5(t_tgelndc5);
tdac.tgdc6(t_tgelndc6); tdac.tgdc7(t_tgelndc7);
```

```
	tdac.tgdacout(t_tgDACout);

	// Tracing waveforms
	sca_util::sca_trace_file *anatf =
	sca_util::sca_create_vcd_trace_file("WaveForms");

	cout << "Start tracing waveforms " << endl;
	sca_util::sca_trace(anatf, t_saw_out, "D(n)_Saw");
	sca_util::sca_trace(anatf, t_sine_out, "D(n)_Sine");
	sca_util::sca_trace(anatf, t_triangle_out, "D(n)_Triangle");
	sca_util::sca_trace(anatf, t_square_out, "Square");
	sca_util::sca_trace(anatf, t_sawthDACout, "S(t)_Saw");
	sca_util::sca_trace(anatf, t_sinDACout, "S(t)_Sine");
	sca_util::sca_trace(anatf, t_tgDACout, "S(t)_Triangle");

	//Simulation period
	sc_start(100, SC_US);

	sca_util::sca_close_vcd_trace_file(anatf);
	cout << "Finished tracing waveforms" << endl;
	return (0);
}
```